# Stochastic Systems

# Runtime Monitors for Markov Decision Processes

Sebastian Junges(✉), Hazem Torfah, and Sanjit A. Seshia

University of California at Berkeley, Berkeley, USA
sjunges@berkeley.edu

**Abstract.** We investigate the problem of monitoring partially observable systems with nondeterministic and probabilistic dynamics. In such systems, every state may be associated with a risk, e.g., the probability of an imminent crash. During runtime, we obtain partial information about the system state in form of observations. The monitor uses this information to estimate the risk of the (unobservable) current system state. Our results are threefold. First, we show that extensions of state estimation approaches do not scale due the combination of nondeterminism and probabilities. While exploiting a geometric interpretation of the state estimates improves the practical runtime, this cannot prevent an exponential memory blowup. Second, we present a tractable algorithm based on model checking conditional reachability probabilities. Third, we provide prototypical implementations and manifest the applicability of our algorithms to a range of benchmarks. The results highlight the possibilities and boundaries of our novel algorithms.

## 1 Introduction

Runtime assurance is essential in the deployment of safety-critical (cyber-physical) systems [12,29,45,49,50]. Monitors observe system behavior and indicate when the system is at risk to violate system specifications. A critical aspect in developing reliable monitors is their ability to handle noisy or missing data. In cyber-physical systems, monitors observe the system state via sensors, i.e., sensors are an interface between the system and the monitor. A monitor has to base its decision solely on the obtained sensor output. These sensors are not perfect, and not every aspect of a system state can be measured.

This paper considers a model-based approach to the construction of monitors for systems with imprecise sensors. Consider Fig. 1(b). We assume a model for the environment together with the controller. Typically, such a model contains both nondeterministic and probabilistic behavior, and thus describes a Markov decision process (MDP): In particular, the sensor is a stochastic process [56] that

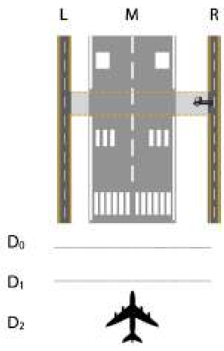
This work is partially supported by NSF grants 1545126 (VeHICaL), 1646208 and 1837132, by the DARPA contracts FA8750-18-C-0101 (AA) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, and by Toyota under the iCyPhy center.

translates the environment state into an observation. For example, this could be a perception module on a plane that during landing estimates the movements of an on-ground vehicle, as depicted in Fig. 1(a). Due to lack of precise data, the vehicle movements itself may be most accurately described using nondeterminism.

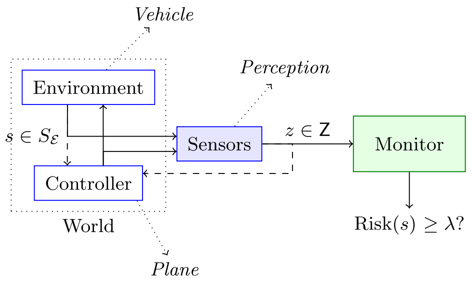
We are interested in the associated *state risk* of the current system state. The state risk may encode, e.g., the probability that the plane will crash with the vehicle within a given number of steps, or the expected time until reaching the other side of the runway. The challenge is that the monitor cannot directly observe the current system state. Instead, the monitor must infer from a trace of observations the current state risk. This cannot be done perfectly as the system state cannot be inferred precisely. Rather, we want a sound, conservative estimate of the system state. More concretely, for a fixed resolution of the nondeterminism, the *trace risk* is the weighted sum over the probability of being in a state having observed the trace, times the risk imposed by this state. The monitoring problem is to decide whether for any possible scheduler resolving the nondeterminism the trace risk of a given trace exceeds a threshold.

Monitoring of systems that contain either only probabilistic or only nondeterministic behavior is typically based on *filtering*. Intuitively, the monitor then estimates the current system states based on the model. For purely nondeterministic systems (without probabilities) a set of states needs to be tracked, and purely probabilistic systems (without nondeterminism) require tracking a distribution over states. This tracking is rather efficient. For systems that contain both probabilistic and nondeterministic behavior, filtering is more challenging. In particular, we show that filtering on MDPs results in an exponential memory blowup as the monitor must track sets of distributions. We show that a reduction based on the geometric interpretation of these distributions is essential for practical performance, but cannot avoid the worst-case exponential blowup. As a tractable alternative to filtering, we rephrase the monitoring problem as the computation of conditional reachability probabilities [9]. More precisely, we unroll and transform the given MDP, and then model check this MDP. This alternative approach yields a polynomial-time algorithm. Indeed, our experiments show the feasibility of computing the risk by computing conditional probabilities. We also show benchmarks on which filtering is a competitive option.

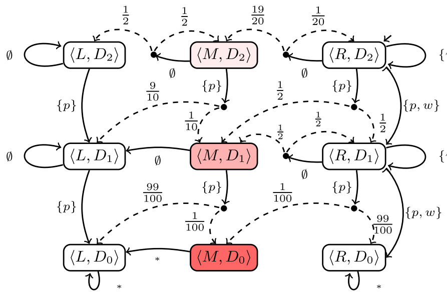
**Contribution and Outline.** This paper presents the first runtime monitoring for systems that can be adequately abstracted by a combination of *probabilities and nondeterminism* and where the system state is *partially observable*. We describe the use case, show that typical filtering approaches in general fail to deal with this setting, and show that a tractable alternative solution exists. In Sect. 3, we investigate *forward filtering*, used to estimate the possible system states in partially observable settings. We show that this approach is tractable for systems that have probabilistic *or* nondeterministic uncertainty, but not for systems that have both. To alleviate the blowup, Sect. 4 discusses an (often) efficacious pruning strategy and its limitations. In Sect. 5 we consider model checking as a more tractable alternative. This result utilizes constructions from the analysis



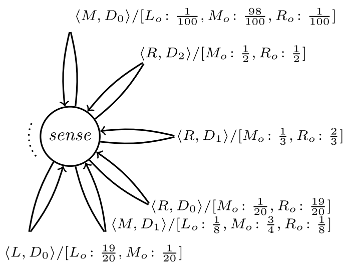
(a) Landing plane scenario



(b) Sensor-based Monitors



(c) World Model


$\langle M, D_0\rangle/[L_o\colon \frac{1}{100}, M_o\colon \frac{98}{100}, R_o\colon \frac{1}{100}]$
$\langle R, D_2\rangle/[M_o\colon \frac{1}{2}, R_o\colon \frac{1}{2}]$
sense
$\langle R, D_1\rangle/[M_o\colon \frac{1}{3}, R_o\colon \frac{2}{3}]$
$\langle R, D_0\rangle/[M_o\colon \frac{1}{20}, R_o\colon \frac{19}{20}]$
$\langle M, D_1\rangle/[L_o\colon \frac{1}{8}, M_o\colon \frac{3}{4}, R_o\colon \frac{1}{8}]$
$\langle L, D_0\rangle/[L_o\colon \frac{19}{20}, M_o\colon \frac{1}{20}]$

(d) Partial Sensor Model

**Fig. 1.** A probabilistic world and sensor model represented by two MDPs for the scenario of an airplane in landing approach with on-ground vehicle movements.

of *partially observable MDPs* and model checking MDPs with *conditional properties*. In Sect. 6 we present baseline implementations of these algorithms, on top of the open-source model checker STORM, and evaluate their performance. The results show that the implementation allows for monitoring of a variety of MDPs, and reveals both strengths and weaknesses of both algorithms. We start with a motivating example and review related work at the end of the paper.

**Motivating Example.** Consider a scenario where an autonomous airplane is in its final approach, i.e., lined up with a designated runway and descending for landing, see Fig. 1(a). On the ground, close to the runway, maintenance vehicles may cross the runway. The airplane tracks the movements of these vehicles and has to decide, depending on the movements of the vehicles, whether to abort the landing. To simplify matters, assume that the airplane (P) is tracking the movement of one vehicle (V) that is about to cross the runway. Let us further assume that P tracks V using a perception module that can only determine the position of the vehicle with a certain accuracy [33], i.e., for every position of V, the perception module reports a noisy variant of the position of V. However, it is important to know that the plane obtains a sequence of these measurements.

Figure 1 illustrates the dynamics of the scenario. The world model describing the movements of V and P is given in Fig. 1(c), where $D_2, D_1$, and $D_0$ define how close P is to the runway, and $R$, $M$, and $L$ define the position of V. Depending on what information V perceives about P, given by the atomic proposition $\{(p)rogress\}$, and what commands it receives $\{(w)ait\}$, it may or may not cross the runway. The perception module receives the information about the state of the world and reports with a certain accuracy (given as a probability) the position of V. The (simple) model of the perception module is given in Fig. 1(d). For example, if P is in zone $D_2$ and V is in $R$ then there is high chance that the perception module returns that V is on the runway. The probability of incorrectly detecting V's position reduces significantly when P is in $D_0$.

A monitor responsible for making the decision to land or to perform a go-around based on the information computed by the perception module, must take into consideration the accuracy of this returned information. For example, if the sequence of sensor readings passed to the monitor is the sequence $\tau = R_o{\cdot}R_o{\cdot}M_o$, and each state is mapped to a certain risk, then how risky is it to land after seeing $\tau$? If, for instance, the world is with high probability in state $\langle M, D_0\rangle$, a very risky state, then the plane should go around. In the paper, we address the question of computing the risk based on this observation sequence. We will use this example as our running example.

## 2 Monitoring with Imprecise Sensors

In this section, we formalize the problem of monitoring with imprecise sensors when both the world and sensor models are given by MDPs. We start with a recap of MDPs, define the monitoring problem for MDPs, and finally show how the dynamics of the system under inspection can be modeled by an MDP defined by the composition of two MDPs of the sensors and world model of the system.

### 2.1 Markov Decision Processes

For a countable set $X$, let $\mathsf{Distr}(X) \subset (X \to [0,1])$ define the set of all distributions over $X$, i.e., for $d \in \mathsf{Distr}(X)$ it holds that $\Sigma_{x\in X} d(x) = 1$. For $d \in \mathsf{Distr}(X)$, let the *support* of $d$ be defined by $\mathsf{supp}(d) := \{x \mid d(x) > 0\}$. We call a distribution $d$ *Dirac*, if $|\mathsf{supp}(d)| = 1$.

**Definition 1 (Markov decision process).** *A* Markov decision process *is a tuple* $\mathcal{M} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$*, where* $S$ *is a finite set of* states*,* $\iota \in \mathsf{Distr}(S)$ *is an* initial distribution*,* $\mathsf{Act}$ *is a finite set of* actions*,* $P\colon S \times \mathsf{Act} \to \mathsf{Distr}(S)$ *is a* partial transition function*,* $\mathsf{Z}$ *is a finite set of* observations*, and* $\mathsf{obs}\colon S \to \mathsf{Distr}(\mathsf{Z})$ *is a* observation function*.*

*Remark 1.* The observation function can also be defined as a state-action observation function $\mathsf{obs}\colon S \times \mathsf{Act} \to \mathsf{Distr}(\mathsf{Z})$. MDPs with state-action observation function can be easily transformed into equivalent MDPs with a state observation function using auxiliary states [19]. Throughout the paper we use state-action observations to keep (sensor) models concise.

For a state $s \in S$, we define $\mathsf{AvAct}(s) = \{\alpha \mid P(s,\alpha) \neq \bot\}$. W.l.o.g., $|\mathsf{AvAct}(s)| \geq 1$. If all distributions in $\mathcal{M}$ are Dirac, we refer to $\mathcal{M}$ as a *Kripke structure* (KS). If $|\mathsf{AvAct}(s)| = 1$ for all $s \in S$, we refer to $\mathcal{M}$ as a *Markov chain* (MC). When $\mathsf{Z} = S$, we refer to $\mathcal{M}$ as *fully observable* and omit $\mathsf{Z}$ and $\mathsf{obs}$ from its definition. A *finite path* in an MDP $\mathcal{M}$ is a sequence $\pi = s_0 a_0 s_1 \ldots s_n \in S \times \left(\mathsf{Act} \times S\right)^*$ such that for every $0 \leq i < n$ it holds that $P(s_i, a_i)(s_{i+1}) > 0$ and $\iota(s_0) > 0$. We denote the set of finite paths of $\mathcal{M}$ by $\Pi_{\mathcal{M}}$. The *length* of the path is given by the number of actions along the path. The set $\Pi^n_{\mathcal{M}}$ for some $n \in \mathbb{N}$ denotes the set of finite paths of length $n$. We use $\pi_{\downarrow}$ to denote the last state in $\pi$. We omit $\mathcal{M}$ whenever it is clear from the context. A *trace* is a sequence of observations $\tau = z_0 \ldots z_n \in \mathsf{Z}^+$. Every path induces a distribution over traces.

As standard, any nondeterminism is resolved by means of a scheduler.

**Definition 2 (Scheduler).** *A* scheduler *for an MDP $\mathcal{M}$ is a function $\sigma \colon \Pi_{\mathcal{M}} \to \mathsf{Distr}(\mathsf{Act})$ with $\mathsf{supp}(\sigma(\pi)) \subseteq \mathsf{AvAct}(\pi_{\downarrow})$ for every $\pi \in \Pi_{\mathcal{M}}$.*

We use $\mathit{Sched}(\mathcal{M})$ to denote the set of schedulers. For a fixed scheduler $\sigma \in \mathit{Sched}(\mathcal{M})$, the probability $\mathsf{Pr}_{\sigma}(\pi)$ of a path $\pi$ (under the scheduler $\sigma$) is the product of the transition probabilities in the induced Markov chain. For more details we refer the reader to [8].

## 2.2 Formal Problem Statement

Our goal is to determine the risk that a system is exposed to having observed a trace $\tau \in \mathsf{Z}^+$. Let $r \colon S \to \mathbb{R}_{\geq 0}$ map states in $\mathcal{M}$ to some risk in $\mathbb{R}_{\geq 0}$. We call $r$ a *state-risk function* for $\mathcal{M}$. This function maps to the risk that is associated with being in every state. For example, in our experiments, we flexibly define the state risk using the (expected reward extension of the) temporal logic PCTL [8], to define the probability of reaching a fail state. For example, we can define risk as the probability to crash within $H$ steps. The use of expected rewards allows for even more flexible definitions.

Intuitively, to compute this risk of the system we need to determine the current system state having observed $\tau$, considering both the probabilistic and nondeterministic context. To this end, we formalize the (conditional) probabilities and risks of paths and traces. Let $\mathsf{Pr}_{\sigma}(\pi \mid \tau)$ define the probability of a path $\pi$, under a scheduler $\sigma$, having observed $\tau$. Since a scheduler may define many paths that induce the observation trace $\tau$, we are interested in the weighted risk over all paths, i.e., $\sum_{\pi \in \Pi^{|\tau|}_{\mathcal{M}}} \mathsf{Pr}_{\sigma}(\pi \mid \tau) \cdot r(\pi_{\downarrow})$. The monitoring problem for MDPs then conservatively over-approximates the risk of a trace by assuming an adversarial scheduler, that is, by taking the supremum risk estimate over all schedulers[1].

[1] We later see in Lemma 8 that this is indeed a maximum.

**The Monitoring Problem.** Given an MDP $\mathcal{M}$, a state-risk $r\colon S \to \mathbb{R}_{\geq 0}$, an observation trace $\tau \in \mathsf{Z}^+$, and a threshold $\lambda \in [0, \infty)$, decide $R_r(\tau) > \lambda$, where the *weighted risk function* $R_r\colon \mathsf{Z}^+ \to \mathbb{R}_{\geq 0}$ is defined as

$$R_r(\tau) \quad := \quad \sup_{\sigma \in Sched(\mathcal{M})} \sum_{\pi \in \Pi_{\mathcal{M}}^{|\tau|}} \mathsf{Pr}_\sigma(\pi \mid \tau) \cdot r(\pi_\downarrow).$$

The conditional probability $\mathsf{Pr}_\sigma(\pi \mid \tau)$ can be characterized using Bayes' rule[2]:

$$\mathsf{Pr}_\sigma(\pi \mid \tau) = \frac{\mathsf{Pr}(\tau \mid \pi) \cdot \mathsf{Pr}_\sigma(\pi)}{\mathsf{Pr}_\sigma(\tau)}.$$

The probability $\mathsf{Pr}(\tau \mid \pi)$ of a trace $\tau$ for a fixed path $\pi$ is $\mathsf{obs}_{\mathsf{tr}}(\pi)(\tau)$, where

$$\mathsf{obs}_{\mathsf{tr}}(s) := \mathsf{obs}(s), \quad \mathsf{obs}_{\mathsf{tr}}(\pi\alpha s') := \{\tau \cdot z \mapsto \mathsf{obs}_{\mathsf{tr}}(\pi)(\tau) \cdot \mathsf{obs}(s')(z)\},$$

when $|\pi| = |\tau|$, and $\mathsf{obs}_{\mathsf{tr}}(\pi)(\tau) = 0$ otherwise. The probability $\mathsf{Pr}_\sigma(\tau)$ of a trace $\tau$ is $\sum_\pi \mathsf{Pr}_\sigma(\pi) \cdot \mathsf{Pr}(\tau \mid \pi)$.

We call the special variant with $\lambda = 0$ the *qualitative monitoring problem.* The problems are (almost) equivalent on Kripke structures, where considering a single path to an adequate state suffices. Details are given in [36, Appendix].

**Lemma 1.** *For Kripke structures the monitoring and qualitative monitoring problems are logspace interreducible.*

In the next sections we present two types of algorithms for the monitoring problem. The first algorithm is based on the widespread (forward) filtering approach [44]. The second is new algorithm based on model checking conditional probabilities. While filtering approaches are efficacious in a purely nondeterministic or a purely probabilistic setting, it does not scale on models such as MDPs that are both probabilistic and nondeterministic. In those models, model checking provides a tractable alternative. Before going into details, we first connect the problem statement more formally to our motivating example.

## 2.3 An MDP Defining the System Dynamics

We show how the weighted risk for a system given by a world and sensor model can be formalized as a monitoring problem for MDPs. To this end, we define the dynamics of the world and sensors that we use as basis for our monitor as the following joint MDP.

For a fully observable world MDP $\mathcal{E} = \langle S_\mathcal{E}, \iota_\mathcal{E}, \mathsf{Act}_\mathcal{E}, P_\mathcal{E}\rangle$ and a sensor MDP $\mathcal{S} = \langle S_\mathcal{S}, \iota_\mathcal{S}, S_\mathcal{E}, P_\mathcal{S}, \mathsf{Z}, \mathsf{obs}\rangle$, where $\mathsf{obs}$ is state-action based, the *inspected system* is defined by an MDP $[\![\langle \mathcal{E}, \mathcal{S}\rangle]\!] = \langle S_\mathcal{J}, \iota_\mathcal{J}, \mathsf{Act}_\mathcal{E}, P_\mathcal{J}, \mathsf{Z}, \mathsf{obs}_\mathcal{J}\rangle$ being the synchronous composition of $\mathcal{E}$ and $\mathcal{S}$:

[2] For conciseness we assume throughout the paper that $\frac{0}{0} = 0$.

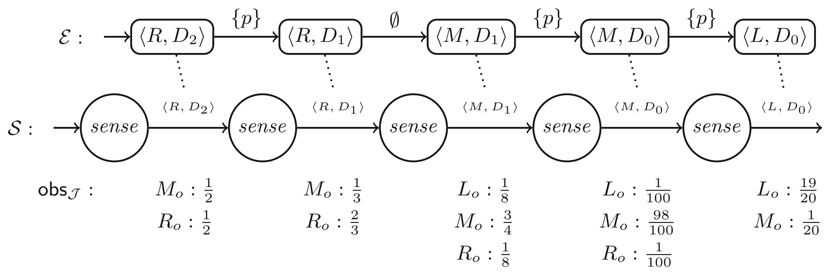


**Fig. 2.** A run with its observations of the inspected system $[\![\langle \mathcal{E}, \mathcal{S}\rangle]\!]$ where $\mathcal{E}$ and $\mathcal{S}$ are the models given in Fig. 1.

- $S_{\mathcal{J}} := S_{\mathcal{E}} \times S_{\mathcal{S}}$,
- $\iota_{\mathcal{J}}$ is defined as $\iota_{\mathcal{J}}(\langle u, s\rangle) := \iota_{\mathcal{E}}(u) \cdot \iota_{\mathcal{S}}(s)$ for each $u \in S_{\mathcal{E}}$ and $s \in S_{\mathcal{S}}$,
- $P_{\mathcal{J}} \colon S_{\mathcal{J}} \times \mathsf{Act}_{\mathcal{E}} \to \mathsf{Distr}(S_{\mathcal{J}})$ such that for all $\langle u, s\rangle \in S_{\mathcal{J}}$ and $\alpha \in \mathsf{Act}_{\mathcal{E}}$;

$$P_{\mathcal{J}}(\langle u, s\rangle, \alpha) = d_{u,s} \in \mathsf{Distr}(S_{\mathcal{J}}),$$

  where for all $u' \in S_{\mathcal{E}}$ and $s' \in S_{\mathcal{S}}$: $d_{u,s}(\langle u', s'\rangle) = P_{\mathcal{E}}(u, \alpha)(u') \cdot P_{\mathcal{S}}(s, u)(s')$,
- $\mathsf{obs}_{\mathcal{J}} \colon S_{\mathcal{J}} \to \mathsf{Distr}(\mathsf{Z})$ with $\mathsf{obs}_{\mathcal{J}} \colon \langle u, s\rangle \mapsto \mathsf{obs}(s, u)$.

In Fig. 2 we illustrate a run of $[\![\langle \mathcal{E}, \mathcal{S}\rangle]\!]$ for the world and sensor MDPs presented in Fig. 1. We particularly show the observations of the joint MDP given by the distributions over the observations for each transition in the run (we omitted the probabilistic transitions for simplicity). The observations of the MDP $\mathcal{M}$ present the output of the sensor upon a path through $\mathcal{M}$. These observations in turn are the inputs to a monitor on top of the system. The role of the monitor is then to compute the risk of being in a critical state based on the received observations.

## 3 Forward Filtering for State Estimation

We start by showing why standard forward filtering does not scale well on MDPs. We briefly show how filtering can be used to solve the monitoring problem for purely nondeterministic systems (Kripke structures) or purely probabilistic systems (Markov Chains). Then, we show why for MDPs, the forward filtering needs to manage, although finite but an exponential set of distributions. In Sect. 4 we present a new improved variant of forward filtering for MDPs based on filtering with vertices of the convex hull. In Sect. 5 we present a new polynomial-time model checking-based algorithm for solving the problem.

### 3.1 State Estimators for Kripke Structures.

For Kripke structures, we maintain a set of possible states that agree with the observed trace. This set of states is inductively characterized by the function

$\mathsf{est}_{\mathsf{KS}}\colon \mathsf{Z}^+ \to 2^S$ which we define formally below. For an observation trace $\tau$, $\mathsf{est}_{\mathsf{KS}}(\tau)$ defines the set of states that can be reached with positive probability. This set can be computed by a forward state traversal [31]. To illustrate how $\mathsf{est}_{\mathsf{KS}}(\tau)$ is computed for $\tau$, consider the underlying Kripke structure of the inspected system $[\![\langle \mathcal{E}, \mathcal{S}\rangle]\!]$ for our running example in Fig. 1 (to make this a Kripke structure, we remove the probabilities). Consider further the observation trace $\tau = R_o \cdot M_o \cdot L_o$. Since $[\![\langle \mathcal{E}, \mathcal{S}\rangle]\!]$ has only one initial state $\langle\langle R, D_2\rangle, \mathit{sense}\rangle$ and $R_o$ is observable with a positive probability in this state, $\mathsf{est}_{\mathsf{KS}}(R_o) = \{\langle\langle R, D_2\rangle, \mathit{sense}\rangle\}$. As $M_o$ is observed next, $\mathsf{est}_{\mathsf{KS}}(R_o \cdot M_o)$ computes the states reached from $\langle\langle R, D_2\rangle, \mathit{sense}\rangle$ and where $M_o$ can be observed with a positive probability, i.e., $\mathsf{est}_{\mathsf{KS}}(R_o \cdot M_o) = \{\langle\langle R, D_1\rangle, \mathit{sense}\rangle, \langle\langle R, M_1\rangle, \mathit{sense}\rangle\}$. Finally, the current state having observed $R_o \cdot M_o \cdot L_o$ may be one of the states $\mathsf{est}_{\mathsf{KS}}(\tau) = \{\langle\langle M, D_1\rangle, \mathit{sense}\rangle,\ \langle\langle L, D_1\rangle, \mathit{sense}\rangle,\ \langle\langle L, D_0\rangle, \mathit{sense}\rangle,\ \langle\langle M, D_0\rangle, \mathit{sense}\rangle\}$, which especially shows that we might be in the high-risk world state $\langle M, D_0\rangle$.

**Definition 3 ($\mathsf{KS}$ state estimator).** *For $\mathsf{KS} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$, the state estimation function $\mathsf{est}_{\mathsf{KS}}\colon \mathsf{Z}^+ \to 2^S$ is defined as*

$$\mathsf{est}_{\mathsf{KS}}(z) := \{s \in S \mid \iota(s) > 0 \wedge \mathsf{obs}(s)(z) > 0\}$$
$$\mathsf{est}_{\mathsf{KS}}(\tau \cdot z) := \Big\{s' \in S \mid \exists s \in \mathsf{est}_{\mathsf{KS}}(\tau), \exists \alpha \in \mathsf{Act}, P(s,\alpha)(s') > 0 \wedge \mathsf{obs}(s')(z) > 0\Big\}.$$

For a Kripke structure $\mathsf{KS}$ and a given trace $\tau$, the monitoring problem can be solved by computing $\mathsf{est}_{\mathsf{KS}}(\tau)$, using [31] and Lemma 1.

**Lemma 2.** *For a Kripke stucture $\mathsf{KS} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$, a trace $\tau \in \mathsf{Z}^+$, and a state-risk function $r\colon S \to \mathbb{R}_{\geq 0}$, it holds that $R_r(\tau) = \max\limits_{s \in \mathsf{est}_{\mathsf{KS}}(\tau)} r(s)$. Computing $R_r(\tau)$ requires time $\mathcal{O}(|\tau| \cdot |P|)$ and space $\mathcal{O}(|S|)$.*

A proof can be found in [36, Appendix]. The time and space requirements follow directly from the inductive definition of $\mathsf{est}_{\mathsf{KS}}$ which resembles solving a forward state traversal problem in automata [31]. In particular, the algorithm allows updating the result after extending $\tau$ in $\mathcal{O}(|P|)$.

### 3.2 State Estimators for Markov Chains

For Markov chains, in addition to tracking the potential reachable system states, we also need to take the transition probabilities into account. When a system is (observation-)deterministic, we can adapt the notion of beliefs, similar to RVSE [54], and similar to the construction of belief MDPs for *partially observable MDPs*, cf. [53]:

**Definition 4 (Belief).** *For an MDP $\mathcal{M}$ with a set of states $S$, a belief $\mathsf{bel}$ is a distribution in $\mathsf{Distr}(S)$.*

In the remainder of the paper, we will denote the function $S \to \{0\}$ by $\mathbf{0}$ and the set $\mathsf{Distr}(S) \cup \{\mathbf{0}\}$ by $\mathsf{Bel}$. A state estimator based on $\mathsf{Bel}$ is then defined as follows [51,54,57][3]:

[3] For the deterministic case, we omit the unique action for brevity.

**Definition 5 (MC state estimator).** *For* $\mathsf{MC} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$, *a trace* $\tau \in \mathsf{Z}^+$ *the state estimation function* $\mathsf{est}_{\mathsf{MC}} \colon \mathsf{Z}^+ \to \mathsf{Bel}$ *is defined as*

$$\mathsf{est}_{\mathsf{MC}}(z) := \begin{cases} \left\{ s \mapsto \frac{\iota(s)\cdot \mathsf{obs}(s)(z)}{\sum\limits_{\hat{s}\in S} \iota(\hat{s})\cdot \mathsf{obs}(\hat{s})(z)} \right\} & \exists s \in S.\ \iota(s)\cdot \mathsf{obs}(z) > 0, \\ \mathbf{0} & \textit{otherwise.} \end{cases}$$

$$\mathsf{est}_{\mathsf{MC}}(\tau\cdot z) := \left\{ s' \mapsto \frac{\sum\limits_{s\in S} \mathsf{est}_{\mathsf{MC}}(\tau)(s)\cdot P(s,s')\cdot \mathsf{obs}(s')(z)}{\sum\limits_{s\in S} \mathsf{est}_{\mathsf{MC}}(\tau)(s)\cdot \big( \sum\limits_{\hat{s}\in S} P(s,\hat{s})\cdot \mathsf{obs}(\hat{s})(z)\big)} \right\}$$

To illustrate how $\mathsf{est}_{\mathsf{MC}}$ is computed, consider again our system in Fig. 1 and assume that the MDP has only the actions labeled with $\{p\}$ (reducing it to the Markov chain induced by the a scheduler that only performs the $\{p\}$ actions). Again we consider the observation trace $\tau = R_o \cdot M_o \cdot L_o$ and compute $\mathsf{est}_{\mathsf{MC}}(\tau)$. For the first observation $R_o$, and since there is only one initial state, it follows that $\mathsf{est}_{\mathsf{MC}}(R_o) = \{\langle R, D_2\rangle \mapsto 1\}$[4]. From $\langle R, D_2\rangle$ and having observed $M_o$ we can reach the states $\langle R, D_1\rangle$ and $\langle M, D_1\rangle$ with probabilities $\mathsf{est}_{\mathsf{MC}}(R_o \cdot M_o) = \{\langle R, D_1\rangle \mapsto \frac{\frac{1}{2}\cdot\frac{1}{3}}{\frac{1}{2}\cdot\frac{1}{3}+\frac{1}{2}\cdot\frac{3}{4}} = \frac{4}{13}, \langle M, D_1\rangle \mapsto \frac{\frac{1}{2}\cdot\frac{3}{4}}{\frac{1}{2}\cdot\frac{1}{3}+\frac{1}{2}\cdot\frac{3}{4}} = \frac{9}{13}\}$. Finally, from the later two states, when observing $L_o$, the states $\langle M, D_0\rangle$ and $\langle L, D_0\rangle$ can be reached with probabilities $\mathsf{est}_{\mathsf{MC}}(R_o \cdot M_o \cdot L_o) = \{\langle M, D_0\rangle \mapsto 0.0001, \langle L, D_0\rangle \mapsto 0.999\}$. Notice that although the state $\langle R, D_0\rangle$ can be reached from $\langle R, D_1\rangle$, the probability of being in this state is 0 since the probability of observing $L_o$ in this state is $\mathsf{obs}(\langle R, D_0\rangle)(L_o) = 0$.

**Lemma 3.** *For a Markov chain* $\mathsf{MC} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$, *a trace* $\tau \in \mathsf{Z}^+$, *and a state-risk function* $r\colon S \to \mathbb{R}_{\geq 0}$, *it holds that* $R_r(\tau) = \sum_{s\in S} \mathsf{est}_{\mathsf{MC}}(\tau)(s)\cdot r(s)$. *Computing* $R_r(\tau)$ *can be done in time* $\mathcal{O}(|\tau|\cdot|S|\cdot|P|)$ *, and using* $|S|$ *many rational numbers. The size of the rationals*[5] *may grow linearly in* $\tau$.

*Proof Sketch.* Since the system is deterministic, there is a unique scheduler $\sigma$, thus $R_r(\tau) = \sum_{\pi\in \Pi^{|\tau|}_{\mathsf{MC}}} \mathsf{Pr}_\sigma(\pi \mid \tau)\cdot r(\pi_\downarrow)$ by definition. We can show by induction over the length of $\tau$ that $\mathsf{Pr}_\sigma(\pi \mid \tau) = \mathsf{est}_{\mathsf{MC}}(\tau)(\pi_\downarrow)$ and conclude that $R_r(\tau) = \sum_{\pi\in \Pi^{|\tau|}_{\mathcal{M}}} \mathsf{est}_{\mathsf{MC}}(\tau)(\pi_\downarrow)\cdot r(\pi_\downarrow) = \sum_{s\in S} \mathsf{est}_{\mathsf{MC}}(\tau)(s)\cdot r(s)$ because $\mathsf{est}_{\mathsf{MC}}(\tau)(s) = 0$ for all $s \in S$ for which there is no path $\pi \in \Pi^{|\tau|}_{\mathcal{M}}$ with $\pi_\downarrow = s$. The complexity follows from the inductive definition of $\mathsf{est}_{\mathsf{MC}}$ that requires in each inductive step to iterate over all transitions of the system and maintain a belief over the states of the system. □

## 3.3 State Estimators for Markov Decision Processes

In an MDP, we have to account for every possible resolution of nondeterminism, which means that a belief can evolve into a set of beliefs:

[4] We omit the (single) sensor state for conciseness.

[5] To avoid growth, one may use fixed-precision numbers that over-approximate the probability of being in any state—inducing a growing (but conservative) error.

**Definition 6 (MDP state estimator).** *For an MDP $\mathcal{M} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$, a trace $\tau \in \mathsf{Z}^+$, and a state-risk function $r\colon S \to \mathbb{R}_{\geq 0}$, the state estimation function $\mathsf{est}_{\mathsf{MDP}}\colon \mathsf{Z}^+ \to 2^{\mathsf{Bel}}$ is defined as*

$$\begin{aligned}\mathsf{est}_{\mathsf{MDP}}(z) &= \{\mathsf{est}_{\mathsf{MC}}(z)\},\\ \mathsf{est}_{\mathsf{MDP}}(\tau \cdot z) &= \left\{\mathsf{bel}' \in \mathsf{Bel} \;\middle|\; \exists \mathsf{bel} \in \mathsf{est}_{\mathsf{MDP}}(\tau).\ \mathsf{bel}' \in \mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathsf{bel}, z)\right\},\end{aligned}$$

*and where $\mathsf{bel}' \in \mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathsf{bel}, z)$ if there exists $\varsigma_{\mathsf{bel}}\colon S \to \mathsf{Distr}(\mathsf{Act})$ such that:*

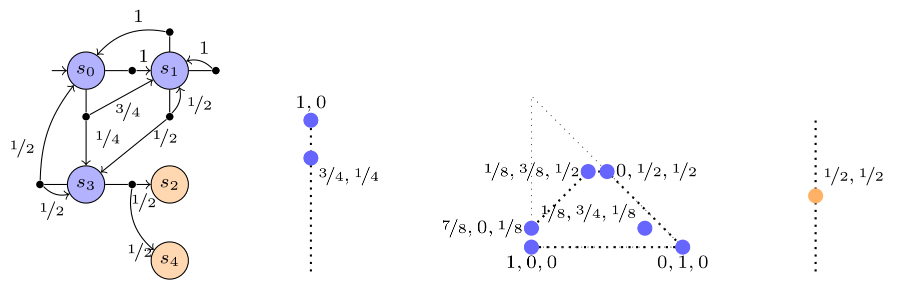
$$\forall s'.\mathsf{bel}'(s') = \frac{\sum\limits_{s\in S} \mathsf{bel}(s) \cdot \sum\limits_{\alpha\in\mathsf{Act}} \varsigma_{\mathsf{bel}}(s)(\alpha) \cdot P(s,\alpha,s') \cdot \mathsf{obs}(s')(z)}{\sum\limits_{s\in S} \mathsf{bel}(s) \cdot \sum\limits_{\alpha\in\mathsf{Act}} \varsigma_{\mathsf{bel}}(s)(\alpha) \cdot \sum\limits_{\hat{s}\in S} P(s,\alpha,\hat{s}) \cdot \mathsf{obs}(\hat{s})(z)}.$$

The definition conservatively extends both Definition 3 and Definition 5. Furthermore, we remark that we do not restrict how the nondeterminism is resolved: any distribution over actions can be chosen, and the distributions may be different for different traces.

Consider our system in Fig. 1. For the trace $\tau = R_o \cdot M_o \cdot L_o$, $\mathsf{est}_{\mathsf{MDP}}(\tau)$ is computed as follows. First, when observing $R_o$, the state estimator computes the initial belief set $\mathsf{est}_{\mathsf{MDP}}(R_o) = \{\{\langle R, D_2\rangle \mapsto 1\}\}$. From this set of beliefs, when observing $M_o$, a set $\mathsf{est}_{\mathsf{MDP}}(R_o \cdot M_o)$ can be computed since all transitions $\emptyset, \{p\}, \{w\}, \{p, w\}$ (as well as their convex combinations) are possible from $\langle R, D_2\rangle$. One of these beliefs is for example $\{\langle R, D_1\rangle \mapsto \frac{4}{13}, \langle M, D_1\rangle \mapsto \frac{9}{13}\}$ when a scheduler takes the transition $\{p\}$ (as was computed in our example for the Markov chain case). Having additionally observed $L_o$ a new set $\mathsf{est}_{\mathsf{MDP}}(R_oM_oL_o)$ of beliefs can be computed based on the beliefs in $\mathsf{est}_{\mathsf{MDP}}(R_oM_o)$. For example from the belief $\{\langle R, D_1\rangle \mapsto \frac{4}{13}, \langle M, D_1\rangle \mapsto \frac{9}{13}\}$, two of the new beliefs are $\{\langle L, D_0\rangle \mapsto 0.999, \langle M, D_0\rangle \mapsto 0.0001\}$ and $\{\langle M, D_1\rangle \mapsto 0.0287, \langle M, D_0\rangle \mapsto 0.0001, \langle L, D_0\rangle \mapsto 0.9712\}$. The first belief is reached by a scheduler that takes a transition $\{p\}$ at both $\langle R, D_1\rangle$ and $\langle M, D_1\rangle$. Notice that the belief does not give a positive probability to the state $\langle R, D_0\rangle$ because $L_o$ cannot be observed in this state. The second belief is reached by considering a scheduler that takes transition $\{p\}$ at $\langle M, D_1\rangle$ and transition $\emptyset$ at $\langle R, D_1\rangle$.

**Theorem 1.** *For an MDP $\mathcal{M} = \langle S, \iota, \mathsf{Act}, P, \mathsf{Z}, \mathsf{obs}\rangle$, a trace $\tau \in \mathsf{Z}^+$, and a state-risk function $r\colon S \to \mathbb{R}_{\geq 0}$, it holds that $R_r(\tau) = \sup_{\mathsf{bel}\in\mathsf{est}_{\mathsf{MDP}}(\tau)} \sum_{s\in S} \mathsf{bel}(s) \cdot r(s)$.*

*Proof Sketch.* For a given trace $\tau$, each (history-dependent, randomizing) scheduler induces a belief over the states of the Markov chain induced by the scheduler. Also, each belief in $\mathsf{est}_{\mathsf{MDP}}(\tau)$ corresponds to a fixed scheduler, namely that one used to compute the belief recursively (i.e., an arbitrary randomizing memoryless scheduler for every time step). Once a scheduler $\sigma$ and its corresponding belief $\mathsf{bel}$ is fixed, or vice versa, we can show using induction over the length of $\tau$ that $\sum_{\pi\in\Pi^{|\tau|}_{\mathcal{M}}} \mathsf{Pr}_\sigma(\pi \mid \tau) \cdot r(\pi_\downarrow) = \sum_{s\in S} \mathsf{bel}(s) \cdot r(s)$. □



(a) MDP $\mathcal{M}$ (b) Over $s_1, s_3$ (c) Over $s_0, s_1, s_3$ (d) Over $s_2, s_3$

**Fig. 3.** Beliefs in $\mathbb{R}^n$ on $\mathcal{M}$ for $\tau = z_0z_0$, $z_0z_0z_0$ and $z_0z_0z_1$, respectively.

## 4 Convex Hull-Based Forward Filtering

In this section, we show that we can use a finite representation for $\mathsf{est}_{\mathsf{MDP}}(\tau)$, but that this representation is exponentially large for some MDPs.

### 4.1 Properties of $\mathsf{est}_{\mathsf{MDP}}(\tau)$.

First, observe that $\mathbf{0}$ never maximizes the risk. Furthermore, $\mathbf{0}$ is closed under updates, i.e., $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathbf{0}, z) = \{\mathbf{0}\}$. We can thus w.l.o.g. assume that $\mathbf{0} \notin \mathsf{est}_{\mathsf{MDP}}(\tau)$. Second, observe that $\mathsf{est}_{\mathsf{MDP}}(\tau) \neq \emptyset$ if $\mathsf{Pr}_\sigma(\tau) > 0$.

We can interpret a belief $\mathsf{bel} \in \mathsf{Bel}$ as point in (a bounded subset of) $\mathbb{R}^{(|S|-1)}$. We are in particular interested in convex sets of beliefs. A set $B \subseteq \mathsf{Bel}$ is convex if the convex hull $\mathsf{CH}(B)$ of $B$, i.e. all convex combination of beliefs in $B$[6], coincides with $B$, i.e., $\mathsf{CH}(B) = B$. For a set $B \subseteq \mathsf{Bel}$, a belief $\mathsf{bel} \in B$ is an interior belief if it can be expressed as convex combination of the beliefs in $B \setminus \{\mathsf{bel}\}$. All other beliefs are (extremal) points or *vertices*. Let the set $\mathcal{V}(B) \subseteq B$ denote the set of *vertices of the convex hull* of $B$.

*Example 1.* Consider Fig. 3(a). All observation are Dirac, and only states $s_2$ and $s_4$ have observation $z_1$. The beliefs having observed $z_0z_0$ are distributions over $s_1, s_3$, and can thus be depicted in a one-dimensional simplex. In particular, we have $\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(z_0z_0)) = \{\{s_1 \mapsto 1\}, \{s_1 \mapsto 3/4, s_3 \mapsto 1/4\}\}$, as depicted in Fig. 3(b). The six beliefs having observed $z_0z_0z_0$ are distributions over $s_0, s_1, s_3$, depicted in Fig. 3(c). Five out of six beliefs are vertices. The belief having observed $z_0z_0z_1$ is in Fig. 3(d).

*Remark 2.* Observe that we illustrate the beliefs over only the states $\mathsf{est}_{\mathsf{KS}}(\tau)$. We therefore call $|\mathsf{est}_{\mathsf{KS}}(\tau)|$ the dimension of $\mathsf{est}_{\mathsf{MDP}}(\tau)$.

From the fundamental theorem of linear programming [47, Ch. 7] it immediately follows that the trace risk $R_\tau$ is obtained at a vertex of the beliefs of $\mathsf{est}_{\mathsf{MDP}}\tau$. We obtain the following refinement over Theorem 1:

[6] That is, $\mathsf{CH}(B) = \{\sum_{\mathsf{bel} \in B} w(\mathsf{bel}) \cdot \mathsf{bel} \mid \text{for all } w \in \mathbb{R}^B_{\geq 0} \text{ with } \sum w(\mathsf{bel}) = 1\}$.

**Theorem 2.** *For every $\tau$ and $r$: $R_r(\tau) = \max\limits_{\mathsf{bel} \in \mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))} \sum_{s \in S} \mathsf{bel}(s) \cdot r(s)$.*

Lemma 5 below clarifies that this maximum indeed exists.

We make some observations that allow us to compute the vertices more efficiently: Let $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)$ denote $\bigcup_{\mathsf{bel} \in B} \mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathsf{bel}, z)$. From the properties of convex sets [18, Ch. 2], we make the following observations: If $B$ is convex, $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)$ is convex, as all operations in computing a new belief are convex-set preserving[7]. Furthermore, if $B$ has a finite set of vertices, then $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)$ has a finite set of vertices. The following lemma which is based on the observations above clarifies how to compute the vertices:

**Lemma 4.** *For a convex set of beliefs $B$ with a finite set of vertices and an observation $z$:*

$$\mathcal{V}(\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)) = \mathcal{V}(\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(\mathcal{V}(B), z)).$$

By induction and using the facts above we obtain:

**Lemma 5.** *Any $\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))$ is finite.*

A monitor thus only needs to track the vertices. Furthermore, $\mathsf{est}^{\mathsf{up}}_{\mathsf{MDP}}(B, z)$ can be adapted to compute only vertices by limiting $\varsigma_{\mathsf{bel}}$ to $S \to \mathsf{Act}$.

## 4.2 Exponential Lower Bounds on the Relevant Vertices

We show that a monitor in general cannot avoid an exponential blow-up in the beliefs it tracks. First observe that updating $\mathsf{bel}$ yields up to $\prod_s |\mathsf{Act}(s)|$ new beliefs (vertex or not), a prohibitively large number. The number of vertices is also exponential:

**Lemma 6.** *There exists a family of MDPs $\mathcal{M}_n$ with $2n + 1$ states such that $|\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))| = 2^n$ for every $\tau$ with $|\tau| > 2$.*

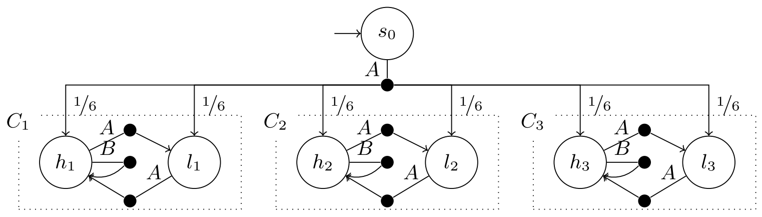
*Proof Sketch.* We construct $\mathcal{M}_n$ with $n = 3$, that is, $\mathcal{M}_3$ in Fig. 4(a). For this MDP and $\tau = AAA$, $|\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))| = 2^3$. In particular, observe how the belief factorizes into a belief within each component $C_i = \{h_i, l_i\}$ and notice that $\mathcal{M}_n$ has components $C_1$ to $C_n$. In particular, for each component, the belief being that we are with probability mass $1/n$ (for $n = 3$, $1/3$) in the 'low' state $l_i$ or the 'high' state $h_i$. We depict the beliefs in Fig. 4(b,c,d). Thus, for any $\tau$ with $|\tau| > 2$ we can compactly represent $\mathcal{V}(\mathsf{est}_{\mathsf{MDP}}(\tau))$ as bit-strings of length $n$. Concretely, the belief

$$\{h_1, l_2, l_3 \mapsto 1/3, l_1, h_2, h_3 \mapsto 0\} \text{ maps to } 100, \text{ and}$$
$$\{h_1, l_2, h_3 \mapsto 1/3, l_1, h_2, l_3 \mapsto 0\} \text{ maps to } 101.$$

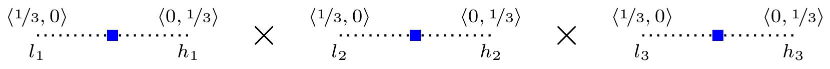
These are exponentially many beliefs for bit strings of length $n$. □

One might ask whether a symbolic encoding of an exponentially large set may result in a more tractable approach to filtering. While Theorem 2 allows

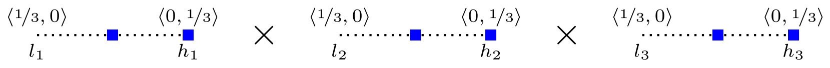
[7] The scaling is called a *projection*.



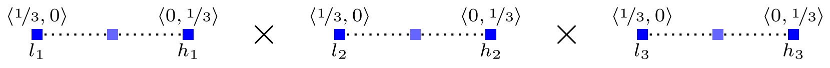
(a) $\mathcal{M}_3$. Observations $\mathsf{Z} = \mathsf{Act}$ with $\mathsf{obs}(s, \alpha) = \alpha$ if $\alpha \neq B$ and $\mathsf{obs}(s, B) = A$ for every $s$. Initial belief is $A$, all probabilities are 1, unless stated otherwise.



(b) Beliefs after $AA$



(c) Beliefs after $AAA$



(d) Beliefs after $AAAA^+$

**Fig. 4.** Construction for the correctness of Lemma 6.

to compute the associated risk from a set of linear constraints with standard techniques, it is not clear whether the concise set of constraints can be efficiently constructed and updated in every step. We leave this concern for future work.

In the remainder we investigate whether we need to track all these beliefs. First, when the monitor is unaware of the state-risk, this is trivially unavoidable. More precisely, all vertices may induce the maximal weighted trace risk by choosing an appropriate state-risk:

**Lemma 7.** *For every $\tau$ and every $\mathsf{bel} \in \mathcal{V}(\mathsf{est_{MDP}}(\tau))$ there exists an $r$ s.t.*

$$\sum_{s \in S} \mathsf{bel}(s) \cdot r(s) \geq \max_{\mathsf{bel}' \in \mathcal{V}(\mathsf{est_{MDP}}(\tau)) \setminus \{\mathsf{bel}\}} \sum_{s \in S} \mathsf{bel}'(s) \cdot r(s) \text{ with } \max_{\mathsf{bel} \in \emptyset} = -\infty.$$

*Proof Sketch.* We construct $r$ such that $r(s) > r(s')$ if $\mathsf{bel}(s) > \mathsf{bel}(s')$. □

Second, even if the monitor is aware of the state risk $r$, it may not be able to prune enough vertices to avoid exponential growth. The crux here is that while some of the current beliefs may induce a smaller risk, an extension of the trace may cause the belief to evolve into a belief that induces the maximal risk.

**Theorem 3.** *There exist MDPs $\mathcal{M}_n$ a $\tau$ with $B := \mathcal{V}(\mathsf{est_{MDP}}(\tau))$ and a state-risk $r$ such that $|B| = 2^n$ and for all $\mathsf{bel} \in B$ exists $\tau' \in \mathsf{Z}^+$ with $R_r(\tau \cdot \tau') > \sup_{\mathsf{bel} \in B'} \sum_s \mathsf{bel}(s) \cdot r(s)$, where $B' = \mathsf{est^{up}_{MDP}}(B \setminus \{\mathsf{bel}\}, \tau')$.*

It is helpful to understand this theorem as describing the outcome of a game between monitor and environment: The statement says if the monitor decides to drop some vertices from $\mathsf{est}_{\mathsf{MDP}}\tau$, the environment may produce an observation trace $\tau'$ that will lead the monitor to underestimate the weighted risk at $R_r(\tau{\cdot}\tau')$.

*Proof Sketch.* We extend the construction of Fig. 4(a) with choices to go to a final state. The full proof sketch can be found in [36, Appendix].

## 4.3 Approximation by Pruning

Finally, we illustrate that we cannot simply prune small probabilities from beliefs. This indicates that an approximative version of filtering for the monitoring problem is nontrivial. Reconsider observing $z_0 z_0$ in the MDP of Fig. 3, and, for the sake of argument, let us prune the (small) entry $s_3 \mapsto 1/4$ to 0. Now, continuing with the trace $z_0 z_0 z_1$, we would update the beliefs from before and then conclude that this trace cannot be observed with positive probability. With pruning, there is no upper bound on the difference between the *computed* $R_\tau$ and the *actual* $R_\tau$. Thus, forward filtering is, in general, not tractable on MDPs.

## 5 Unrolling with Model Checking

We present a tractable algorithm for the monitoring problem. Contrary to filtering, this method incorporates the state risk. We briefly consider the qualitative case. An algorithm that solves that problem iteratively guesses a successor such that the given trace has positive probability, and reaches a state with sufficient risk. The algorithm only stores the current and next state and a counter.

**Theorem 4.** *The Monitoring Problem with $\lambda = 0$ is in NLOGSPACE.*

This result implies the existence of a polynomial time algorithm, e.g., using a graph-search on a graph growing in $|\tau|$. There also is a deterministic algorithm with space complexity $\mathcal{O}(log^2(|\mathcal{M}|+|\tau|))$, which follows from applying Savitch's Theorem [46] , but that algorithm has exponential time complexity.

We now present a tractable algorithm for the quantitative case, where we need to store all paths. We do this efficiently by storing an unrolled MDP with these paths using ideas from [9,19]. In particular, on this MDP, we can efficiently obtain the scheduler that optimizes the risk by model checking rather than enumerating over all schedulers explicitly. We give the result before going into details.

**Theorem 5.** *The Monitoring Problem (with $\lambda > 0$) is P-complete.*

The problem is P-hard, as unary-encoded step-bounded reachability is P-hard [41]. It remains to show a P-time algorithm[8], which is outlined below. Roughly, the algorithm constructs an MDP $\mathcal{M}'''$ from $\mathcal{M}$ in three conceptual steps, such that the

[8] On first sight, this might be surprising as step-bounded reachability in MDPs is PSPACE-hard and only quasi-polynomial. However, our problem gets a trace and therefore (assuming that the trace is not compressed) can be handled in time polynomial in the length of the trace.

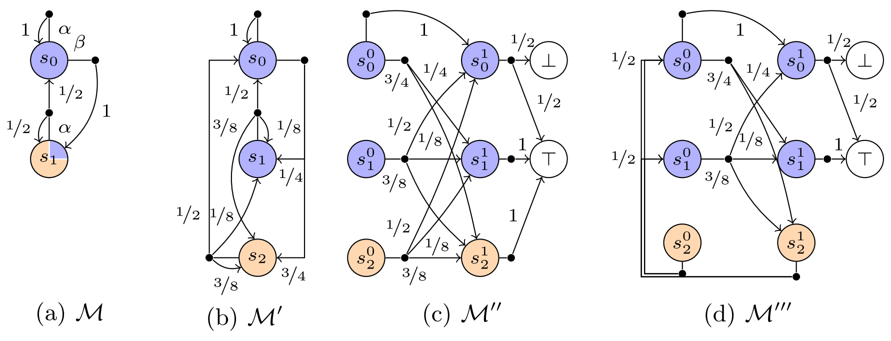


**Fig. 5.** Polynomial-time algorithm for solving Problem 1 illustrated.

maximal probability of reaching a state in $\mathcal{M}'''$ coincides with the $R_r(\tau)$. The former can be solved by linear programming in polynomial time. The downside is that even in the best case, the memory consumption grows linearly in $|\tau|$.

We outline the main steps of the algorithm and exemplify them below: First, we transform $\mathcal{M}$ into an MDP $\mathcal{M}'$ with *deterministic state* observations, i.e., with $\mathsf{obs}' \colon S \to \mathsf{Z}$. This construction is detailed in [19, Remark 1], and runs in polynomial time. The new initial distribution takes into account the initial observation and the initial distribution. Importantly, for each path $\pi$ and each trace $\tau$, $\mathsf{obs}_{\mathsf{tr}}(\pi)(\tau)$ is preserved. From here, the idea for the algorithm is a tailored adaption of the construction for conditional reachability probabilities in [9]. We ensure that $r(s) \in [0,1]$ by scaling $r$ and $\lambda$ accordingly. Now, we construct a new MDP $\mathcal{M}'' = \langle S'', \iota'', \mathsf{Act}'', P'' \rangle$ with state space $S'' := (S' \times \{0,\dots,|\tau|-1\}) \cup \{\bot, \top\}$ and an $n$-times unrolled transition relation. Furthermore, from the states $\langle s, |\tau|-1 \rangle$, there is a single outgoing action that with probability $r(s)$ leads to $\top$ and with probability $1 - r(s)$ leads to $\bot$. Observe that the risk is now the supremum of conditioned reachability probabilities over paths that reach $\top$, conditioned by the trace $\tau$. The MDP $\mathcal{M}''$ is only polynomially larger. Then, we construct MDP $\mathcal{M}'''$ by copying $\mathcal{M}''$ and replacing (part of) the transition relation $P''$ by $P'''$ such that paths $\pi$ with $\tau \notin \mathsf{obs}_{\mathsf{tr}}(\pi)$ are looped back to the initial state (resembling rejection sampling). Formally,

$$P'''(\langle s, i \rangle, \alpha) = \begin{cases} P''(\langle s, i \rangle, \alpha) & \text{if } \mathsf{obs}'(s) = \tau_i, \\ \iota & \text{otherwise.} \end{cases}$$

The maximal conditional reachability probability in $\mathcal{M}''$ is the maximal reachability probability in $M'''$ [9]. Maximal reachability probabilities can be computed by solving a linear program [43], and can thus be computed in polynomial time.

*Example 2.* We illustrate the construction in Fig. 5. In Fig. 5(a), we depict an MDP $\mathcal{M}$, with $\iota = \{s_0, s_1 \mapsto 1/2\}$. Furthermore, let $\tau = z_0 z_0$ and let $r(s_0) = 1$ and $r(s_1) = 2$. Let $\mathsf{obs}(s_0) = \{z_0 \mapsto 1\}$ and $\mathsf{obs}(s_1) = \{z_0 \mapsto 1/4, z_1 \mapsto 3/4\}$. State $s_1$ has two possible observations, so we split $s_1$ into $s_1$ and $s_2$ in MDP

$\mathcal{M}'$, each with their own observations. Any transition into $s_1$ is now split. As $|\tau| = 2$, we unroll the MDP $\mathcal{M}'$ into MDP $\mathcal{M}''$ to represent two steps, and add goal and sink states. After rescaling, we obtain that $r(s_0) = 1/2$, whereas $r(s_1) = r(s_2) = 2/2 = 1$, and we add the appropriate outgoing transitions to the states $s_*^1$. In a final step, we create MDP $\mathcal{M}'''$ from $\mathcal{M}''$: we reroute all probability mass that does not agree with the observations to the initial states. Now, $R_r(z_0 z_0)$ is given by the probability to reach, in $\mathcal{M}'''$, in an unbounded number of steps, $\top$.

The construction also implies that maximizing over a finite set of schedulers, namely the deterministic schedulers with a counter from 0 to $|\tau|$, suffices. We denote this class $\Sigma_{\mathrm{DC}}(|\tau|)$. Formally, a scheduler is in $\Sigma_{\mathrm{DC}}(k)$ if for all $\pi, \pi'$:

$$\Big(\pi_\downarrow = \pi'_\downarrow \wedge \big(|\pi| = |\pi'| \vee (|\pi| > k \wedge |\pi'| > k)\big)\Big) \text{ implies } \sigma(\pi) = \sigma(\pi').$$

**Lemma 8.** *For every $\tau$, it holds that*

$$R_r(\tau) \quad = \quad \max_{\sigma \in \Sigma_{DC}(|\tau|)} \sum_{\pi \in \Pi_M^{|\tau|}} \mathsf{Pr}_\sigma(\pi \mid \tau) \cdot r(\pi_\downarrow).$$

The crucial idea underpinning this lemma is that memoryless schedulers suffice for the unrolling, and that the states of the unrolling can be uniquely mapped to a state and the length of the history for every $\pi$ through $\mathcal{M}$. By reducing step-bounded reachability we can also show that this set of schedulers is necessary [4].

## 6 Empirical Evaluation

*Implementation.* We provide prototype implementations for both filtering- and model-checking-based approaches from Sect. 3, built on top of the probabilistic model checker Storm [30]. We provide a schematic setup of our implementation in Fig. 6. As input, we consider a symbolic description of MDPs with state-based observation labels, based on an extended dialect of the Prism language. We define the state risk in this MDP via a temporal property (given as a PCTL formula), and obtain the concrete state-risk by model checking. We take a seed that yields a trace using the simulator. For the experiments, actions are resolved uniformly in this simulator[9]. The simulator iteratively feeds observations into the monitor, running either of our two algorithms (implemented in C++). After each observation $z_i$, the monitor computes the risk $R_i$ having observed $z_0 \dots z_i$. We flexibly combine these components via a Python API[10].

For filtering as in Sect. 4, we provide a sparse data structure for beliefs that is updated using only deterministic schedulers. This is sufficient, see Lemma 4. To further prune the set of beliefs, we implement an SMT-driven elimination [48]

[9] This is not an assumption but rather our evaluation strategy.

[10] Available at https://github.com/monitoring-MDPs/premise.

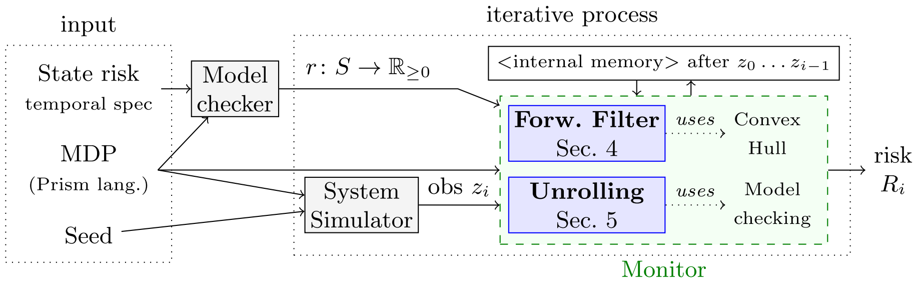


**Fig. 6.** Schematic setup for prototype mapping stream $z_0 \dots z_k$ to stream $R_0 \dots R_k$.

of interior beliefs, inside of the convex hull[11]. We construct the unrolling as described in Sect. 5 and apply model checking via any sparse engines in STORM.

*Reproducibility.* We archived a container with sources, benchmarks, and scripts to reproduce our experiments: https://doi.org/10.5281/zenodo.4724622.

*Set-Up.* For each benchmark described below, we sampled 50 random traces using seeds 0–49 of lengths up to $|\tau| = 500$. We are interested in the *promptness*, that is, the delay of time between getting an observation $z_i$ and returning corresponding risk $r_i$, as well as the *cumulative performance* obtained by summing over the promptness along the trace. We use a timeout of 1 second for this query. We compare the forward filtering (FF) approach with and without convex hull (CH) reduction, and the model unrolling approach (UNR) with two model checking engines of STORM: exact policy iteration (EPI, [43]) and optimistic value iteration (OVI, [28]). All experiments are run on a MacBook Pro MV962LL/A, using a single core. The memory limit of 6GB was not violated. We use Z3 [38] as SMT-solver [11] for the convex hull reduction.

*Benchmarks.* We present three benchmark families, all MDPs with a combination of probabilities, nondeterminism and partial observability.

AIRPORT-A is as in Sect. 1, but with a higher resolution for both ground vehicle in the middle lane and the plane. AIRPORT-B has a two-state sensor model with stochastic transitions between them.

REFUEL-A models robots with a depleting battery and recharging stations. The world model consists of a robot moving around in a $D \times D$ grid with some dedicated charging cells, where each action costs energy. The risk is to deplete the battery within a fixed horizon. REFUEL-B is a two-state sensor variant.

EVADE-I is inspired by a navigation task in a multi-agent setting in a $D \times D$ grid. The monitored robot moves randomly, and the risk is defined as the probability of crashing with the other robot. The other robot has an internal incentive in the form of a cardinal direction, and nondeterministically decides to move or

[11] Advanced algorithms like Quickhull [10] are not without significant adaptions applicable as the set of beliefs can be degenerate (roughly, a set without full rank).

**Table 1.** Performance for promptness of online monitoring on various benchmarks.

| | | | | | | CH Forward Filtering | | | | | | | Unrolling | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Id | Name | Inst | $\lvert S\rvert$ | $\lvert P\rvert$ | $\lvert\tau\rvert$ | $N$ | $T_{\text{avg}}$ | $T_{\text{max}}$ | $B_{\text{avg}}$ | $B_{\text{max}}$ | $D_{\text{avg}}$ | $D_{\text{max}}$ | $N$ | $T_{\text{avg}}$ | $T_{\text{max}}$ | $\lvert S_u\rvert_{\text{avg}}$ | $\lvert S_u\rvert_{\text{max}}$ |
| 1 | AIRPORT-A | 7,50,30 | 20910 | 114143 | 100 | **50** | 0.01 | 0.01 | 4.5 | 7 | 4.6 | 7 | **50** | 0.04 | 0.11 | 524 | 599 |
| | | | | | 500 | **50** | 0.01 | 0.01 | 1.0 | 1 | 1.0 | 1 | **50** | 0.01 | 0.01 | 1075 | 1258 |
| 2 | AIRPORT-B | 3,50,30 | 20232 | 106012 | 100 | **0** | | | | | | | **50** | 0.09 | 0.16 | 556 | 629 |
| | | | | | 500 | **0** | | | | | | | **50** | 0.01 | 0.01 | 1460 | 1647 |
| 3 | AIRPORT-B | 7,50,30 | 41820 | 308474 | 100 | **0** | | | | | | | **50** | 0.14 | 0.33 | 1000 | 1183 |
| | | | | | 500 | **0** | | | | | | | **11** | 0.02 | 0.02 | 2097 | 2297 |
| 4 | REFUEL-A | 12,50 | 45073 | 2431691 | 100 | **50** | 0.01 | 0.01 | 2.2 | 4 | 2.8 | 5 | **50** | 0.01 | 0.05 | 325 | 409 |
| | | | | | 500 | **50** | 0.01 | 0.01 | 1.5 | 4 | 1.7 | 5 | **50** | 0.01 | 0.19 | 1071 | 2409 |
| 5 | REFUEL-B | 12,50 | 90145 | 9725277 | 100 | **50** | 0.06 | 0.23 | 4.2 | 8 | 5.6 | 10 | **50** | 0.04 | 0.17 | 608 | 732 |
| | | | | | 500 | **50** | 0.01 | 0.01 | 2.9 | 8 | 3.3 | 10 | **46** | 0.04 | 0.09 | 2171 | 4688 |
| 6 | EVADE-I | 15 | 377101 | 2022295 | 100 | **50** | 0.01 | 0.02 | 2.6 | 10 | 3.3 | 4 | **49** | 0.01 | 0.06 | 332 | 363 |
| | | | | | 500 | **50** | 0.01 | 0.01 | 2.4 | 5 | 3.4 | 4 | **45** | 0.08 | 0.90 | 1655 | 1891 |
| 7 | EVADE-V | 5,3 | 1001 | 5318 | 100 | **26** | 0.01 | 0.01 | 1.0 | 1 | 1.0 | 1 | **50** | 0.00 | 0.02 | 134 | 241 |
| | | | | | 500 | **25** | 0.01 | 0.01 | 1.0 | 1 | 1.0 | 1 | **50** | 0.00 | 0.01 | 538 | 671 |
| 8 | EVADE-V | 6,3 | 2161 | 11817 | 100 | **1** | 0.01 | 0.01 | 1.0 | 1 | 1.0 | 1 | **50** | 0.02 | 0.32 | 319 | 861 |
| | | | | | 500 | **1** | 0.01 | 0.01 | 1.0 | 1 | 1.0 | 1 | **49** | 0.01 | 0.02 | 777 | 1484 |

to uniformly randomly change its incentive. The monitor observes everything except the incentive of the other robot. EVADE-V is an alternative navigation task: Contrary to above, the other robot does not have an internal state and indeed navigates nondeterministically in one of the cardinal directions. We only observe the other robot location is within the view range.

*Results.* We split our results in two tables. In Table 1, we give an ID for every benchmark name and instance, along with the size of the MDP (nr. of states $|S|$ and transitions $|P|$) our algorithms operate on. We consider the promptness after prefixes of length $|\tau|$. In particular, for forward filtering with the convex hull optimization, we give the number $N$ of traces that did not time out before, and consider the average $T_{\text{avg}}$ and maximal time $T_{\text{max}}$ needed (over all sampled traces that did not time-out before). Furthermore, we give the average, $B_{\text{avg}}$, and maximal, $B_{\text{max}}$, number of beliefs stored (after reduction), and the average, $D_{\text{avg}}$, and maximal, $D_{\text{max}}$, dimension of the belief support. Likewise, for unrolling with exact model checking, we give the number $N$ of traces that did not time out before, and we consider average $T_{\text{avg}}$ and maximal time $T_{\text{max}}$, as well as the average size and maximal number of states of the unfolded MDP.

In Table 2, we consider for the benchmarks above the cumulative performance. In particular, this table also considers an alternative implementation for both FF and UNR. We use the IDs to identify the instance, and sum for each prefix of length $|\tau|$ the time. For filtering, we recall the number of traces $N$ that did not time out, the average and maximal cumulative time along the trace, the average cumulative number of beliefs that were considered, and the average cumulative number of beliefs eliminated. For the case without convex hull, we do not eliminate any vertices. For unrolling, we report average $T_{\text{avg}}$ and maximal cumulative time using EPI, as well as the time required for model building, *Bld*$^{\%}$ (relative to the total time, per trace). We compare this to the average

**Table 2.** Summarized performance for online monitoring

| | | FF w/o CH | | | | FF w/ CH | | | | | UNR (EPI) | | | | | UNR (OVI) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Id | $\lvert\tau\rvert$ | $N$ | $T_{\text{avg}}$ | $T_{\text{max}}$ | $B_{\text{avg}}$ | $N$ | $T_{\text{avg}}$ | $T_{\text{max}}$ | $B_{\text{avg}}$ | $E_{\text{avg}}$ | $N$ | $T_{\text{avg}}$ | $T_{\text{max}}$ | $Bld^{\%}_{\text{avg}}$ | $Bld^{\%}_{\text{max}}$ | N | $T_{\text{avg}}$ | $T_{\text{max}}$ |
| 1 | 100 | **0** | | | | **50** | 0.9 | 1.1 | 493 | 241 | **50** | 2.9 | 3.6 | 6 | 56 | **50** | 0.0 | 0.1 |
| | 500 | **0** | | | | **50** | 3.7 | 4.3 | 1040 | 316 | **50** | 7.5 | 10.7 | 21 | 24 | **50** | 0.4 | 0.8 |
| 2 | 100 | **0** | | | | **0** | | | | | **50** | 3.7 | 4.7 | 6 | 54 | **50** | 0.1 | 0.1 |
| | 500 | **0** | | | | **0** | | | | | **50** | 11.9 | 17.1 | 18 | 23 | **50** | 0.6 | 0.8 |
| 3 | 100 | **0** | | | | **0** | | | | | **50** | 7.6 | 10.6 | 5 | 55 | **50** | 0.1 | 0.2 |
| | 500 | **0** | | | | **0** | | | | | **11** | 21.3 | 28.7 | 19 | 23 | **50** | 0.9 | 1.7 |
| 4 | 100 | **1** | 0.9 | 0.9 | 1473 | **50** | 0.7 | 0.8 | 241 | 138 | **50** | 0.7 | 1.0 | 35 | 69 | **50** | 0.0 | 0.1 |
| | 500 | **1** | 0.9 | 0.9 | 1873 | **50** | 3.4 | 3.7 | 868 | 226 | **50** | 5.6 | 21.2 | 57 | 67 | **50** | 0.5 | 0.9 |
| 5 | 100 | **0** | | | | **50** | 7.4 | 10.7 | 442 | 2267 | **50** | 2.5 | 4.4 | 32 | 57 | **50** | 0.1 | 0.2 |
| | 500 | **0** | | | | **50** | 16.5 | 42.2 | 1781 | 4249 | **46** | 19.5 | 64.2 | 55 | 70 | **50** | 1.3 | 2.3 |
| 6 | 100 | **13** | 0.7 | 2.9 | 2055 | **50** | 1.1 | 4.8 | 273 | 160 | **49** | 0.5 | 2.0 | 34 | 65 | **47** | 0.0 | 0.1 |
| | 500 | **2** | 4.4 | 6.8 | 20524 | **50** | 5.1 | 11.5 | 1237 | 632 | **45** | 22.4 | 53.6 | 13 | 29 | **43** | 0.5 | 0.7 |
| 7 | 100 | **13** | 0.1 | 0.5 | 274 | **26** | 0.8 | 1.2 | 106 | 11 | **50** | 0.4 | 1.0 | 19 | 45 | **48** | 0.0 | 0.1 |
| | 500 | **13** | 0.1 | 0.5 | 674 | **25** | 3.7 | 4.2 | 505 | 7 | **50** | 1.3 | 4.4 | 46 | 58 | **47** | 0.2 | 0.3 |
| 8 | 100 | **0** | | | | **1** | 1.3 | 1.3 | 124 | 109 | **50** | 1.5 | 7.0 | 15 | 39 | **36** | 0.4 | 5.6 |
| | 500 | **0** | | | | **1** | 4.3 | 4.3 | 524 | 109 | **49** | 4.9 | 28.1 | 37 | 56 | **35** | 0.7 | 6.4 |

and maximal cumulative time for using OVI (notice that building times remain approximately the same).

*Discussion.* The results from our prototype show that conservative (sound) predictive modeling of systems that combine probabilities, nondeterminism and partial observability is within reach with the methods we proposed and state-of-the-art algorithms. Both forward filtering and an unrolling-based approaches have their merits. The practical results thus slightly diverge from the complexity results in Sect. 3.1, due to structural properties of some benchmarks. In particular, for AIRPORT-A and REFUEL-A, the nondeterminism barely influences the belief, and so there is no explosion, and consequentially the dimension of the belief is sufficiently small that the convex hull can be efficiently computed. Rather than the number of states, this belief dimension makes EVADE-V a difficult benchmark[12]. *If many states can be reached with a particular trace, and if along these paths there are some probabilistic states, forward filtering suffers significantly.* We see that if the benchmark allows for efficacious forward filtering, it is not slowed down in the way that unrolling is slower on longer traces. For UNR, we observe that OVI is typically the fastest, but EPI does not suffer from the numerical worst-cases as OVI does. *If an observation trace is unlikely, the unrolled MDP constitutes a numerically challenging problem, in particular for value-iteration based model checkers*, see [27]. For FF, the convex hull computation is essential for any dimension, and eliminating some vertices in every step keeps the number of belief states manageable.

[12] The max dimension =1 in EVADE-V is only over the traces that did not time-out. The dimension when running in time-outs is above 5.

## 7 Related Work

We are not the first to consider model-based runtime verification in the presence of partial observability and probabilities. Runtime verification with state estimation on hidden Markov models (HMM)—without nondeterminism has been studied for various types of properties [51,54,57] and has been extended to hybrid systems [52]. The tool Prevent focusses on black-box systems by learning an HMM from a set of traces. The HMM approximates (with only convergence-in-the-limit guarantees) the actual system [6], and then estimates during runtime the most likely trace rather than estimating a distribution over current states. Extensions consider symmetry reductions on the models [7]. These techniques do not make a conservative (sound) risk estimation. The recent framework for runtime verification in the presence of partial observability [23] takes a more strict black-box view and cannot provide state estimates. Finally, [26] chooses to have partial observability to make monitoring of software systems more efficient, and [58] monitors a noisy sensor to reduce energy consumption.

State beliefs are studied when verifying HMMs [59], where the question whether a sequence of observations likely occurs, or which HMM is an adequate representation of a system [37]. State beliefs are prominent in the verification of partially observable MDPs [16,32,40], where one can observe the actions taken (but the problem itself is to find the right scheduler). Our monitoring problem can be phrased as a special case of verification of partially observable stochastic games [20], but automatic techniques for those very general models are lacking. Likewise, the idea of *shielding* (pre)computes all action choices that lead to safe behavior [3,5,15,24,34,35]. For partially observable settings, shielding again requires to compute partial-information schedulers [21,39], contrary to our approach. Partial observability has also been studied in the context of diagnosability, studying if a fault has occurred (in the past) [14], or what actions uncover faults [13]. We, instead assume partial observability in which we do detect faults, but want to estimate the risk that these faults occur in the future.

The assurance framework for reinforcement learning [42] implicitly allows for stochastic behavior, but cannot cope with partial observability or nondeterminism. Predictive monitoring has been combined with deep learning [17] and Bayesian inference [22], where the key problem is that the computation of an imminent failure is too expensive to be done exactly. More generally, learning automata models has been motivated with runtime assurance [1,55]. Testing approaches statistically evaluate whether traces are likely to be produced by a given model [25]. The approach in [2] studies stochastic black-box systems with controllable nondeterminism and iteratively learns a model for the system.

## 8 Conclusion

We have presented the first framework for monitoring based on a trace of observations on models that combine nondeterminism and probabilities. Future work includes heuristics for approximate monitoring and for faster convex hull computations, and to apply this work to gray-box (learned) models.

**Acknowledgments.** The example in Fig. 1 is inspired by a challenge problem provided by Boeing in the DARPA Assured Autonomy program.

## References

1. Aichernig, B.K., et al.: Learning a behavior model of hybrid systems through combining model-based testing and machine learning. In: Gaston, C., Kosmatov, N., Le Gall, P. (eds.) ICTSS 2019. LNCS, vol. 11812, pp. 3–21. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31280-0_1
2. Aichernig, B.K., Tappler, M.: Probabilistic black-box reachability checking (extended version). Formal Methods Syst. Des. **54**(3), 416–448 (2019)
3. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI, pp. 2669–2678. AAAI Press (2018)
4. Andova, S., Hermanns, H., Katoen, J.-P.: Discrete-time rewards model-checked. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 88–104. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-40903-8_8
5. Avni, G., Bloem, R., Chatterjee, K., Henzinger, T.A., Könighofer, B., Pranger, S.: Run-time optimization for learned controllers through quantitative games. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 630–649. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_36
6. Babaee, R., Gurfinkel, A., Fischmeister, S.: *Prevent*: a predictive run-time verification framework using statistical learning. In: Johnsen, E.B., Schaefer, I. (eds.) SEFM 2018. LNCS, vol. 10886, pp. 205–220. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92970-5_13
7. Babaee, R., Gurfinkel, A., Fischmeister, S.: Predictive run-time verification of discrete-time reachability properties in black-box systems using trace-level abstraction and statistical learning. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 187–204. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_11
8. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
9. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 515–530. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_43
10. Barber, C.B., Dobkin, D.P., Huhdanpaa, H.: The Quickhull algorithm for convex hulls. ACM Trans. Math. Softw. **22**(4), 469–483 (1996)
11. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
12. Bartocci, E., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
13. Bertrand, N., Fabre, É., Haar, S., Haddad, S., Hélouët, L.: Active diagnosis for probabilistic systems. In: Muscholl, A. (ed.) FoSSaCS 2014. LNCS, vol. 8412, pp. 29–42. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54830-7_2
14. Bertrand, N., Haddad, S., Lefaucheux, E.: A tale of two diagnoses in probabilistic systems. Inf. Comput. **269** (2019)

15. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: runtime enforcement for reactive systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 533–548. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_51
16. Bork, A., Junges, S., Katoen, J.-P., Quatmann, T.: Verification of indefinite-horizon POMDPs. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 288–304. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_16
17. Bortolussi, L., Cairoli, F., Paoletti, N., Smolka, S.A., Stoller, S.D.: Neural predictive monitoring. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 129–147. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_8
18. Boyd, S.P., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge (2014)
19. Chatterjee, K., Chmelik, M., Gupta, R., Kanodia, A.: Optimal cost almost-sure reachability in POMDPs. Artif. Intell. **234**, 26–48 (2016)
20. Chatterjee, K., Doyen, L.: Partial-observation stochastic games: how to win when belief fails. ACM Trans. Comput. Log. **15**(2), 16:1–16:44 (2014)
21. Chatterjee, K., Novotný, P., Pérez, G.A., Raskin, J., Zikelic, D.: Optimizing expectation with guarantees in POMDPs. In: AAAI, pp. 3725–3732. AAAI Press (2017)
22. Chou, Y., Yoon, H., Sankaranarayanan, S.: Predictive runtime monitoring of vehicle models using Bayesian estimation and reachability analysis. In: IROS (2020, to appear)
23. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification with partial observability and resets. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 165–184. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_10
24. Dräger, K., Forejt, V., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. Log. Methods Comput. Sci. **11**(2) (2015)
25. Gerhold, M., Stoelinga, M.: Model-based testing of probabilistic systems. Formal Asp. Comput. **30**(1), 77–106 (2018)
26. Grigore, R., Kiefer, S.: Selective monitoring. In: CONCUR. LIPIcs, vol. 118, pp. 20:1–20:16. Dagstuhl - LZI (2018)
27. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)
28. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 488–511. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_26
29. Havelund, K., Roşu, G.: Runtime verification - 17 years later. In: Colombo, C., Leucker, M. (eds.) RV 2018. LNCS, vol. 11237, pp. 3–17. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03769-7_1
30. Hensel, C., Junges, S., Katoen, J., Quatmann, T., Volk, M.: The probabilistic model checker storm. CoRR abs/2002.07080 (2020)
31. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Algorithmic analysis of nonlinear hybrid systems. IEEE Trans. Autom. Control **43**(4), 540–554 (1998)
32. Horák, K., Bosanský, B., Chatterjee, K.: Goal-HSVI: heuristic search value iteration for goal POMDPs. In: IJCAI, pp. 4764–4770. ijcai.org (2018)
33. Jansen, N., Humphrey, L., Tumova, J., Topcu, U.: Structured synthesis for probabilistic systems. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2019. LNCS, vol. 11460, pp. 237–254. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_16

34. Jansen, N., Könighofer, B., Junges, S., Serban, A., Bloem, R.: Safe reinforcement learning using probabilistic shields (invited paper). In: CONCUR. LIPIcs, vol. 171, pp. 3:1–3:16. Dagstuhl - LZI (2020)
35. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.-P.: Safety-constrained reinforcement learning for MDPs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 130–146. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_8
36. Junges, S., Torfah, H., Seshia, S.A.: Runtime monitoring for Markov decision processes. CoRR abs/2105.12322 (2021)
37. Kiefer, S., Sistla, A.P.: Distinguishing hidden Markov chains. In: LICS, pp. 66–75. ACM (2016)
38. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
39. Nam, W., Alur, R.: Active learning of plans for safety and reachability goals with partial observability. IEEE Trans. Syst. Man Cybern. Part B **40**(2), 412–420 (2010)
40. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. Real Time Syst. **53**(3), 354–402 (2017)
41. Papadimitriou, C.H., Tsitsiklis, J.N.: The complexity of Markov decision processes. Math. Oper. Res. **12**(3), 441–450 (1987)
42. Phan, D.T., Grosu, R., Jansen, N., Paoletti, N., Smolka, S.A., Stoller, S.D.: Neural simplex architecture. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 97–114. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_6
43. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics. Wiley (1994)
44. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE **77**(2), 257–286 (1989)
45. Sánchez, C., et al.: A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. **54**(3), 279–335 (2019)
46. Savitch, W.J.: Relationships between nondeterministic and deterministic tape complexities. J. Comput. Syst. Sci. **4**(2), 177–192 (1970)
47. Schrijver, A.: Theory of Linear and Integer Programming. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley (1999)
48. Seidel, R.: Convex hull computations. In: Handbook of Discrete and Computational Geometry, 2nd edn, pp. 495–512. Chapman and Hall/CRC (2004)
49. Seshia, S.A.: Introspective environment modeling. In: Finkbeiner, B., Mariani, L. (eds.) RV 2019. LNCS, vol. 11757, pp. 15–26. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32079-9_2
50. Seshia, S.A., Sadigh, D., Sastry, S.S.: Towards verified artificial intelligence. arXiv e-prints, July 2016
51. Sistla, A.P., Srinivas, A.R.: Monitoring temporal properties of stochastic systems. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 294–308. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78163-9_25
52. Sistla, A.P., Žefran, M., Feng, Y.: Runtime monitoring of stochastic cyber-physical systems with hybrid state. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 276–293. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_21

53. Spaan, M.T.J.: Partially observable Markov decision processes. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning, Adaptation, Learning, and Optimization, vol. 12, pp. 387–414. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27645-3_12
54. Stoller, S.D., et al.: Runtime verification with state estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29860-8_15
55. Tappler, M., Aichernig, B.K., Bacci, G., Eichlseder, M., Larsen, K.G.: $L^*$-based learning of Markov decision processes. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 651–669. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_38
56. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. Intelligent Robotics and Autonomous Agents. MIT Press (2005)
57. Wilcox, C.M., Williams, B.C.: Runtime verification of stochastic, faulty systems. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 452–459. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_34
58. Woo, H., Mok, A.K.: Real-time monitoring of uncertain data streams using probabilistic similarity. In: RTSS, pp. 288–300. IEEE CS (2007)
59. Zhang, L., Hermanns, H., Jansen, D.N.: Logic and model checking for hidden Markov models. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 98–112. Springer, Heidelberg (2005). https://doi.org/10.1007/11562436_9

# Model Checking Finite-Horizon Markov Chains with Probabilistic Inference

Steven Holtzen[1(✉)], Sebastian Junges[2], Marcell Vazquez-Chanlatte[2], Todd Millstein[1], Sanjit A. Seshia[2], and Guy Van den Broeck[1]

[1] University of California, Los Angeles, CA, USA
`sholtzen@cs.ucla.edu`
[2] University of California, Berkeley, CA, USA

**Abstract.** We revisit the symbolic verification of Markov chains with respect to finite horizon reachability properties. The prevalent approach iteratively computes step-bounded state reachability probabilities. By contrast, recent advances in probabilistic inference suggest symbolically representing all horizon-length paths through the Markov chain. We ask whether this perspective advances the state-of-the-art in probabilistic model checking. First, we formally describe both approaches in order to highlight their key differences. Then, using these insights we develop Rubicon, a tool that transpiles Prism models to the probabilistic inference tool `Dice`. Finally, we demonstrate better scalability compared to probabilistic model checkers on selected benchmarks. All together, our results suggest that probabilistic inference is a valuable addition to the probabilistic model checking portfolio, with Rubicon as a first step towards integrating both perspectives.

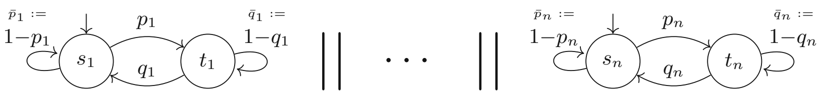
## 1 Introduction

Systems with probabilistic uncertainty are ubiquitous, e.g., probabilistic programs, distributed systems, fault trees, and biological models. Markov chains replace nondeterminism in transition systems with probabilistic uncertainty, and *probabilistic model checking* [4,7] provides model checking algorithms. A key property that probabilistic model checkers answer is: *What is the (precise) probability that a target state is reached (within a finite number of steps h)*? Contrary to classical *qualitative* model checking and approximate variants of probabilistic model checking, precise probabilistic model checking must find the total probability of *all* paths from the initial state to any target state.

S. Holtzen and S. Junges—Contributed equally.

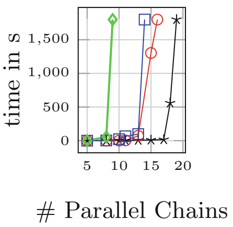
This work is partially supported by NSF grants #IIS-1943641, #IIS-1956441, #CCF-1837129, DARPA grant #N66001-17-2-4032, a Sloan Fellowship, a UCLA Dissertation Year Fellowship, and gifts by Intel and Facebook Research.
This work is partially supported by NSF grants 1545126 (VeHICaL), 1646208 and 1837132, by the DARPA contracts FA8750-18-C-0101 (AA) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, and by Toyota under the iCyPhy center.

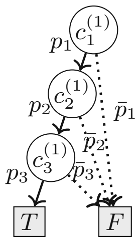
A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 577–601, 2021.
https://doi.org/10.1007/978-3-030-81688-9_27



(a) Motivating factory Markov chain with $s_i = \llbracket c_i = 0 \rrbracket, t_i = \llbracket c_i = 1 \rrbracket$.

```
const double p1, p2, p3, q1, q2, q3;
module F1
 c1  : bool init false;
 [a] !c1 -> p1: (c1'=1) +1-p1: (c1'=0);
 [a]  c1 -> q1: (c1'=0) +1-q1: (c1'=1);
endmodule
module F2 = F1[c1=c2,p1=p2,q1=q2]
module F3 = F1[c1=c3,p1=p3,q1=q3]
label "allStrike" = c1 & c2 & c3;
```





(b) A PRISM model of (a) with 3 factories. (c) Relative scaling. (d) BDD

**Fig. 1.** Motivating example. Figure 1(c) compares the performance of RUBICON (—⋆—), STORM's explicit engine (—○—), STORM's symbolic engine (—□—) and PRISM (—◇—) when invoked on a (b) with arbitrarily fixed (different) constants for $p_i, q_i$ and horizon $h = 10$. Times are in seconds, with a time-out of 30 min.

Nevertheless, the prevalent ideas in probabilistic model checking are generalizations of qualitative model checking. Whereas qualitative model checking tracks the states that can reach a target state (or dually, that can be reached from an initial state), probabilistic model checking tracks the $i$-step reachability probability for each state in the chain. The $i+1$-step reachability can then be computed via multiplication with the *transition matrix*. The scalability concern is that this matrix grows with the state space in the Markov chain. Mature model checking tools such as STORM [36], Modest [34], and PRISM [51] utilize a variety of methods to alleviate the state space explosion. Nevertheless various natural models cannot be analyzed by the available techniques.

In parallel, within the AI community a different approach to representing a distribution has emerged, which on first glance can seem unintuitive. Rather than marginalizing out the paths and tracking reachability probabilities per state, the probabilistic AI community commonly aggregates all *paths* that reach the target state. At its core, inference is then a weighted sum over all these paths [16]. This hinges on the observation that this set of paths can often be stored more compactly, and that the probability of two paths that share the same prefix or suffix can be efficiently computed on this concise representation. This *inference technique* has been used in a variety of domains in the artificial intelligence (AI) and verification communities [9,14,27,39], but is not part of any mature probabilistic model checking tools.

This paper theoretically and experimentally compares and contrasts these two approaches. In particular, we describe and motivate RUBICON, a probabilistic model checker that *leverages the successful probabilistic inference techniques*. We begin with an example that explains the core ideas of RUBICON followed by the paper structure and key contributions.

**Motivating Example.** Consider the example illustrated in Fig. 1(a). Suppose there are $n$ factories. Each day, the workers at each factory collectively decide whether or not to strike. To simplify, we model each factory $(i)$ with two states, striking $(t_i)$ and not striking $(s_i)$. Furthermore, since no two factories are identical, we take the probability to begin striking $(p_i)$ and to stop striking $(q_i)$ to be different for each factory. Assuming that each factory transitions synchronously and in parallel with the others, we query: "what is the probability that all the factories are simultaneously striking within $h$ days?"

Despite its simplicity, we observe that state-of-the-art model checkers like STORM and PRISM do not scale beyond 15 factories.[1] For example, Fig. 1(b) provides a PRISM encoding for this simple model (we show the instance with 3 factories), where a Boolean variable $c_i$ is used to encode the state of each factory. The "`allStrike`" label identifies the target state. Figure 1(c) shows the run time for an increasing number of factories. While all methods eventually time out, RUBICON scales to systems with an order of magnitude more states.

*Why is This Problem Hard?* To understand the issue with scalability, observe that tools such as STORM and PRISM store the transition matrix, either explicitly or symbolically using algebraic decision diagrams (ADDs). Every distinct entry of this transition matrix needs to be represented; in the case of ADDs using a unique leaf node. Because each factory in our example has a different probability of going on strike, that means each subset of factories will likely have a unique probability of jointly going on strike. Hence, the transition matrix then will have a number of distinct probabilities that is exponential in the number of factories, and its representation as an ADD must blow up in size. Concretely, for 10 factories, the size of the ADD representing the transition matrix has 1.9 million nodes. Moreover, the explicit engine fails due to the dense nature of the underlying transition matrix. We discuss this method in Sect. 3.

*How to Overcome This Limitation?* This problematic combinatorial explosion is often unnecessary. For the sake of intuition, consider the simple case where the horizon is 1. Still, the standard transition matrix representations blow up exponentially with the number of factories $n$. Yet, the probability of reaching the "`allStrike`" state is easy to compute, even when $n$ grows: it is $p_1 \cdot p_2 \cdots p_n$.

RUBICON aims to compute probabilities in this compact *factorized* way by representing the computation as a binary decision diagram (BDD). Figure 1(d) gives an example of such a BDD, for three factories and a horizon of one. A key property of this BDD, elaborated in Sect. 3, is that it can be interpreted as a *parametric Markov chain*, where the weight of each edge corresponds with the probability of a particular factory striking. Then, the probability that the goal state is reached is given by the weighted sum of paths terminating in $T$: for this instance, there is a single such path with weight $p_1 \cdot p_2 \cdot p_3$. These BDDs are tree-like Markov-chains, so model checking can be performed in time linear in the size

[1] Section 6 describes the experimental apparatus and our choice of comparisons.

of the BDD using dynamic programming. Essentially, the BDD represents the set of paths that reach a target state—an idea common in probabilistic inference.

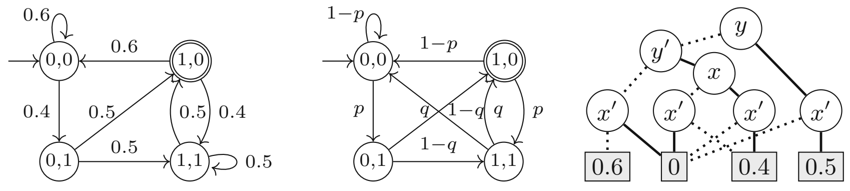
To construct this BDD, we propose to encode our reachability query symbolically as a *weighted model counting* (WMC) query on a logical formula. By compiling that formula into a BDD, we obtain a diagram where computing the query probability can be done efficiently (in the size of the BDD). Concretely for Fig. 1(d), the BDD represents the formula $c_1^{(1)} \land c_2^{(1)} \land c_3^{(1)}$, which encodes all paths through the chain that terminate in the goal state (all factories strike on day 1). For this example and this horizon, this is a single path. WMC is a well-known strategy for probabilistic inference and is currently the among the state-of-the-art approaches for discrete graphical models [16], discrete probabilistic programs [39], and probabilistic logic programs [27].

In general, the exponential growth of the number of paths might seem like it dooms this approach: for $n = 3$ factories and horizon $h = 1$, we need to only represent 8 paths, but for $h = 2$, we would need to consider 64 different paths, and so on. However, a key insight is that, for many systems – such as the factory example – the structural compression of BDDs allows a concise representation of exponentially many paths, all *while* being parametric over path probabilities (see Sect. 4). To see why, observe that in the above discussion, the state of each factory is *independent* of the other factories: independence, and its natural generalizations like *conditional* and *contextual* independence, are the driving force behind many successful probabilistic inference algorithms [47]. Succinctly, the key advantage of Rubicon is that it exploits a form of structure that has thus far been under-exploited by model checkers, which is why it scales to more parallel factories than the existing approaches on the hard task. In Sect. 6 we consider an extension to this motivating example that adds dependencies between factories. This dependency (or rather, the accompanying increase in the size of the underlying MC) significantly decreases scalability for the existing approaches but negligibly affects Rubicon.

This leads to the task: *how does one go from a* Prism *model to a concise BDD efficiently*? To do this, Rubicon leverages a novel translation from Prism models into a probabilistic programming language called `Dice` (outlined in Sect. 5).

**Contribution and Structure.** Inspired by the example, we contribute conceptual and empirical arguments for leveraging BDD-based probabilistic inference in model checking. Concretely:

1. We demonstrate fundamental advantages in using probabilistic inference on a natural class of models (Sect. 1 and 6).
2. We explain these advantages by showing the fundamental differences between existing model checking approaches and probabilistic inference (Sect. 3 and 4). To that end, Sect. 4 presents probabilistic inference based on an operational and a logical perspective and combines these perspectives.
3. We leverage those insights to build Rubicon, a tool that transpiles Prism to `Dice`, a probabilistic programming language (Sect. 5).



(a) Toy-example $\mathcal{M}$ (b) pMC $\mathcal{M}'$ (c) For $\mathcal{M}$: $P$ as ADD

**Fig. 2.** (a) MC toy example (b) (distinct) pMC toy example (c) ADD transition matrix

4. We demonstrate that RUBICON indeed attains an order-of-magnitude scaling improvement on several natural problems including sampling from parametric Markov chains and verifying network protocol stabilization (Sect. 6).

Ultimately we argue that RUBICON makes a valuable contribution to the portfolio of probabilistic model checking backends, and brings to bear the extensive developments on probabilistic inference to well-known model checking problems.

## 2 Preliminaries and Problem Statement

We state the problem formally and recap relevant concepts. See [7] for details. We sometimes use $\bar{p}$ to denote $1-p$. A *Markov chain* (MC) is a tuple $\mathcal{M} = \langle S, \iota, P, T\rangle$ with $S$ a (finite) set of *states*, $\iota \in S$ the *initial state*, $P\colon S \to Distr(S)$ the *transition function*, and $T$ a set of *target states* $T \subseteq S$, where $Distr(S)$ is the set of distributions over a (finite) set $S$. We write $P(s, s')$ to denote $P(s)(s')$ and call $P$ a *transition matrix*. The successors of $s$ are $\mathsf{Succ}(s) = \{s' \mid P(s, s') > 0\}$. To support MCs with billions of states, we may describe MCs symbolically, e.g., with PRISM [51] or as a probabilistic program [42,48]. For such a symbolic description $\mathcal{P}$, we denote the corresponding MC with $[\![\,\mathcal{P}\,]\!]$. States then reflect assignments to symbolic variables.

A *path* $\pi = s_0 \ldots s_n$ is a sequence of states, $\pi \in S^+$. We use $\pi_\downarrow$ to denote the *last state* $s_n$, and the *length* of $\pi$ above is $n$ and is denoted $|\pi|$. Let $\text{Paths}_h$ denote the paths of length $h$. The probability of a path is the product of the transition probabilities, and may be defined inductively by $\Pr(s) = 1$, $\Pr(\pi \cdot s) = \Pr(\pi) \cdot P(\pi_\downarrow, s)$. For a fixed *horizon* $h$ and set of states $T$, let the set $[\![\, s{\to}\Diamond^{\leq h} T \,]\!] = \{\pi \mid \pi_0 = s \wedge |\pi| \leq h \wedge \pi_\downarrow \in T \wedge \forall i < |\pi|.\ \pi_i \notin T\}$ denote paths from $s$ of length at most $h$ that terminate at a state contained in $T$. Furthermore, let $\Pr_{\mathcal{M}}(s \models \Diamond^{\leq h} T) = \sum_{\pi \in [\![\, s \to \Diamond^{\leq h} T \,]\!]} \Pr(\pi)$ describe the probability to reach $T$ within $h$ steps. We simplify notation when $s = \iota$ and write $[\![\, \Diamond^{\leq h} T \,]\!]$ and $\Pr_{\mathcal{M}}(\Diamond^{\leq h} T)$, respectively. We omit $\mathcal{M}$ whenever that is clear from the context.

**Formal Problem:** Given an MC $\mathcal{M}$ and a horizon $h$, compute $\Pr_{\mathcal{M}}(\Diamond^{\leq h} T)$.

*Example 1.* For conciseness, we introduce a toy example MC $\mathcal{M}$ in Fig. 2(a). For horizon $h = 3$, there are three paths that reach state $\langle 1{,}0 \rangle$: For example the path $\langle 0, 0\rangle\langle 0, 1\rangle\langle 1, 0\rangle$ with corresponding reachability probability $0.4 \cdot 0.5$. The reachability probability $\Pr_{\mathcal{M}}(\Diamond^{\leq 3}\{\langle 1, 0\rangle\}) = 0.42$.

It is helpful to separate the topology and the probabilities. We do this by means of a *parametric MC* (pMC) [22]. A pMC over a fixed set of parameters $\boldsymbol{p}$ generalises MCs by allowing for a transition function that maps to $\mathbb{Q}[\boldsymbol{p}]$, i.e., to polynomials over these variables [22]. A pMC and a *valuation* of parameters $\boldsymbol{u} \colon \boldsymbol{p} \to \mathbb{R}$ describe a MC by replacing $\boldsymbol{p}$ with $\boldsymbol{u}$ in the transition function $P$ to obtain $P[\boldsymbol{u}]$. If $P[\boldsymbol{u}](s)$ is a distribution for every $s$, then we call $\boldsymbol{u}$ a *well-defined* valuation. We can then think about a pMC $\mathcal{M}$ as a generator of a set of MCs $\{\mathcal{M}[\boldsymbol{u}] \mid \boldsymbol{u} \text{ well-defined}\}$. Figure 2(b) shows a pMC; any valuation $\boldsymbol{u}$ with $\boldsymbol{u}(p), \boldsymbol{u}(q) \in [0, 1]$ is well-defined. We consider the following associated problem:

**Parameter Sampling:** Given a pMC $\mathcal{M}$, a finite set of well-defined valuations $U$, and a horizon $h$, compute $\Pr_{\mathcal{M}[\boldsymbol{u}]}(\Diamond^{\leq h} T)$ for each $\boldsymbol{u} \in U$.

We recap binary *decision diagrams* (BDDs) and their generalization into algebraic decision diagrams (ADDs, a.k.a. multi-terminal BDDs). ADDs over a set of variables $X$ are directed acyclic graphs whose vertices $V$ can be partitioned into *terminal nodes* $V_t$ without successors and *inner nodes* $V_i$ with two successors. Each terminal node is labeled with a polynomial over some parameters $\boldsymbol{p}$ (or just to constants in $\mathbb{Q}$), $\mathsf{val} \colon V_t \to \mathbb{Q}[\boldsymbol{p}]$, and each inner node $V_i$ with a variable, $\mathsf{var} \colon V_i \to X$. One node is the root node $v_0$. Edges are described by the two successor functions $E_0 \colon V_i \to V$ and $E_1 \colon V_i \to V$. A BDD is an ADD with exactly two terminals labeled $T$ and $F$. Formally, we denote an ADD by the tuple $\langle V, v_0, X, \mathsf{var}, \mathsf{val}, E_0, E_1 \rangle$. ADDs describe functions $f \colon \mathbb{B}^X \to \mathbb{Q}[\boldsymbol{p}]$ (described by a path in the underlying graph and the label of the corresponding terminal node). As finite sets can be encoded with bit vectors, ADDs represent functions from (tuples of) finite sets to polynomials.

*Example 2.* The transition matrix $P$ of the MC in Fig. 2(a) maps states, encoded by bit vectors, $\langle x, y\rangle, \langle x', y'\rangle$ to the probabilities to move from state $\langle x, y\rangle$ to $\langle x', y'\rangle$. Figure 2(c) shows the corresponding ADD.[2]

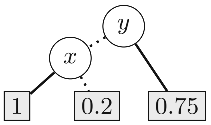
## 3 A Model Checking Perspective

We briefly analyze the de-facto standard approach to symbolic probabilistic model checking of finite-horizon reachability probabilities. It is an adaptation of qualitative model checking, in which we track the (backward) reachable states. This set can be thought of as a mapping from states to a Boolean indicating whether a target state can be reached. We generalize the mapping to a function that maps every state $s$ to the probability that we reach $T$ within $i$ steps,

[2] The ADD also depends on the variable order, which we assume fixed for conciseness.

| state | horizon $h$ | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| ⟨0,0⟩ | 0 | 0 | 0.2 | 0.42 |
| ⟨0,1⟩ | 0 | 0.5 | 0.75 | 0.875 |
| ⟨1,0⟩ | 1 | 1 | 1 | 1 |
| ⟨1,1⟩ | 0 | 0.5 | 0.75 | 0.875 |

(a) $\Pr_{\mathcal{M}}(\Diamond^{\leq h}\{\langle 0,1\rangle\})$



(b) $\Pr_{\mathcal{M}}(\Diamond^{\leq 2}\{\langle 0,1\rangle\})$ as ADD

**Fig. 3.** Bounded reachability and symbolic model checking for the MC $\mathcal{M}$ in Fig. 2(a).

denoted $\Pr_{\mathcal{M}}(s \models \Diamond^{\leq i} T)$. First, it is convenient to construct a transition relation in which the target states have been made absorbing, i.e., we define a matrix with $A(s,s') = P(s,s')$ if $s \notin T$ and $A(s,s') = [s = s']$[3] otherwise. The following *Bellman equations* characterize that aforementioned mapping:

$$\Pr_{\mathcal{M}}\left(s \models \Diamond^{\leq 0} T\right) = [s \in T],$$

$$\Pr_{\mathcal{M}}\left(s \models \Diamond^{\leq i} T\right) = \sum_{s' \in \mathsf{Succ}(s)} A(s,s') \cdot \Pr_{\mathcal{M}}(s' \models \Diamond^{\leq i-1} T) \qquad \text{with } i > 0.$$

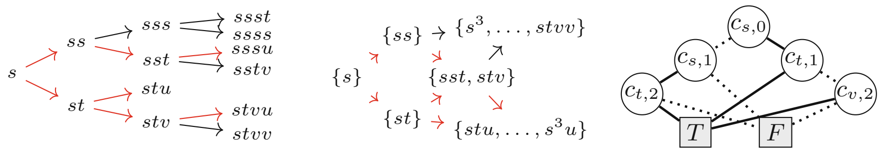
The main aspect model checkers take from these equations is that to compute the $h$-step reachability from state $s$, one only needs to combine the $h{-}1$-step reachability from any state $s'$ *and* the transition probabilities $P(s,s')$. We define a vector $\boldsymbol{T}$ with $\boldsymbol{T}(s) = [s \in T]$. The algorithm then iteratively computes and stores the $i$ step reachability for $i = 0$ to $i = h$, e.g. by computing $A^3 \cdot \boldsymbol{T}$ using $A \cdot (A \cdot (A \cdot \boldsymbol{T}))$. This reasoning is thus *inherently backwards* and *implicitly marginalizing out paths*. In particular, rather than storing the $i$-step paths that lead to the target, one only stores a vector $\boldsymbol{x} = A^i \cdot \boldsymbol{T}$ that stores for every state $s$ the sum over all $i$-long paths from $s$.

Explicit representations of matrix $A$ and vector $\boldsymbol{x}$ require memory at least in the order $|S|$.[4] To overcome this limitation, *symbolic* probabilistic model checking stores both $A$ and $A^i \cdot \boldsymbol{T}$ as an ADD by considering the matrix as a function from a tuple $\langle s,s'\rangle$ to $A(s,s')$, and $\boldsymbol{x}$ as a function from $s$ to $\boldsymbol{x}(s)$ [2].

*Example 3.* Reconsider the MC in Fig. 2(a). The $h$-bounded reachability probability $\Pr_{\mathcal{M}}(\Diamond^{\leq h}\{\langle 1,0\rangle\})$ can be computed as reflected in Fig. 3(a). The ADD for $P$ is shown in Fig. 2(c). The ADD for $\boldsymbol{x}$ when $h = 2$ is shown in Fig. 3(b).

The performance of symbolic probabilistic model checking is directly governed by the sizes of these two ADDs. The size of an ADD is bounded from below by the number of leafs. In qualitative model checking, both ADDs are in fact BDDs, with two leafs. However, for the ADD representing $A$, this lower bound is given by the number of different probabilities in the transition matrix. In the running example, we have seen that a small program $\mathcal{P}$ may have an underlying MC $[\![\mathcal{P}]\!]$ with an exponential state space $S$ and equally many different transition probabilities. Symbolic probabilistic model checking also scales

[3] Where $[x]$=1 if $x$ holds and 0 otherwise.
[4] Excluding e.g., partial exploration or sampling which typically are not exact.



(a) $\mathsf{CT}(\mathcal{M}, 3)$ (b) $\mathsf{CT}(\mathcal{M}, 3)$ compressed (c) Predicate as BDD

**Fig. 4.** The computation tree for $\mathcal{M}$ and horizon 3 and its compression. We label states as $s{=}\langle 0{,}0\rangle$, $t{=}\langle 0{,}1\rangle$, $u{=}\langle 1{,}0\rangle$, $v{=}\langle 1{,}1\rangle$. Probabilities are omitted for conciseness.

badly on some models where $A$ has a concise encoding but $\boldsymbol{x}$ has too many different entries.[5] Therefore, model checkers may store $\boldsymbol{x}$ partially explicit [49].

The insights above are not new. Symbolic probabilistic model checking has advanced [46] to create small representations of both $A$ and $\boldsymbol{x}$. In competitions, Storm often applies a bisimulation-to-explicit method that extracts an explicit representation of the bisimulation quotient [26,36]. Finally, game-based abstraction [32,44] can be seen as a predicate abstraction technique on the ADD level. However, these methods do not change the computation of the finite horizon reachability probabilities and thus do not overcome the inherent weaknesses of the iterative approach in combination with an ADD-based representation.

## 4 A Probabilistic Inference Perspective

We present four key insights into probabilistic inference. **(1)** Sect. 4.1 shows how probabilistic inference takes the classical definition as summing over the set of paths, and turns this definition into an algorithm. In particular, these paths may be stored in a computation tree. **(2)** Sect. 4.2 gives the traditional reduction from probabilistic inference to the classical weighted model counting (WMC) problem [16,57]. **(3)** Sect. 4.3 connects this reduction to point (1) by showing that a BDD that represents this WMC is *bisimilar* to the computation tree assuming that the out-degree of every state in the MC is two. **(4)** Sect. 4.4 describes and compares the computational benefits of the BDD representation. In particular, we clarify that enforcing an out-degree of two is a key ingredient to overcoming one of the weaknesses of symbolic probabilistic model checking: the number of different probabilities in the underlying MC.

### 4.1 Operational Perspective

The following perspective frames (an aspect of) probabilistic inference as a model transformation. By definition, the set of all paths – each annotated with the transition probabilities – suffices to extract the reachability probability. These sets of paths may be represented in the computation tree (which is itself an MC).

[5] For an interesting example of this, see the "Queue" example in Sect. 6.

*Example 4.* We continue from Example 1. We put all paths of length three in a computation tree in Fig. 4(a) (cf. the caption for state identifiers). The three paths that reach the target are highlighted in red. The MC is highly redundant. We may compress to the MC in Fig. 4(b).

**Definition 1.** *For MC $\mathcal{M}$ and horizon $h$, the computation tree (CT) $\mathsf{CT}(\mathcal{M}, h) = \langle \mathit{Paths}_h, \iota, P', T' \rangle$ is an MC with states corresponding to paths in $\mathcal{M}$, i.e., $\mathit{Paths}_h^{\mathcal{M}}$, initial state $\iota$, target states $T' = [\![ \Diamond^{\leq h} T ]\!]$, and transition relation*

$$P'(\pi, \pi') = \begin{cases} P(\pi_{\downarrow}, s) & \text{if } \pi_{\downarrow} \notin T \wedge \pi' = \pi.s, \\ [\pi_{\downarrow} \in T \wedge \pi' = \pi] & \text{otherwise.} \end{cases} \tag{1}$$

The CT contains (up to renaming) the same paths to the target as the original MC. Notice that after $h$ transitions, all paths are in a sink state, and thus we can drop the step bound from the property and consider either finite or indefinite horizons. The latter considers all paths that eventually reach the target. We denote the probability mass of these paths with $\mathrm{Pr}_{\mathcal{M}}(s \models \Diamond T)$ and refer to [7] for formal details.[6] Then, we may compute bounded reachability probabilities in the original MC by analysing unbounded reachability in the CT:

$$\mathrm{Pr}_{\mathcal{M}}(\Diamond^{\leq h} T) = \mathrm{Pr}_{\mathsf{CT}(\mathcal{M},h)}(\Diamond^{\leq h} T') = \mathrm{Pr}_{\mathsf{CT}(\mathcal{M},h)}(\Diamond T').$$

The nodes in the CT have a natural topological ordering. The unbounded reachability probability is then computed (efficiently in CT's size) using dynamic programming (i.e., topological value iteration) on the Bellman equation for $s \notin T$:

$$\mathrm{Pr}_{\mathcal{M}}(s \models \Diamond T) = \textstyle\sum_{s' \in \mathsf{Succ}(s)} P(s, s') \cdot \mathrm{Pr}_{\mathcal{M}}(s' \models \Diamond T).$$

For pMCs, the right-hand side naturally is a factorised form of the *solution function* $f$ that maps parameter values to the induced reachability probability, i.e. $f(\boldsymbol{u}) = \mathrm{Pr}_{\mathcal{M}[\boldsymbol{u}]}(\Diamond^{\leq h} T)$ [22,24,33]. For bounded reachability (or acyclic pMCs), this function amounts to a sum over all paths with every path reflected by a term of a polynomial, i.e., the sum is a polynomial. In sum-of-terms representation, the polynomial can be exponential in the number of parameters [5].

For computational efficiency, we need a smaller representation of the CT. As we only consider reachability of $T$, we may simplify [43] the notion of (weak) bisimulation [6] (in the formulation of [40]) to the following definition.

**Definition 2.** *For $\mathcal{M}$ with states $S$, a relation $\mathcal{R} \subseteq S \times S$ is a (weak) bisimulation (with respect to $T$) if $s\mathcal{R}s'$ implies $\mathrm{Pr}_{\mathcal{M}}(s \models \Diamond T) = \mathrm{Pr}_{\mathcal{M}}(s' \models \Diamond T)$. Two states $s, s'$ are (weakly) bisimilar (with respect to $T$) if $\mathrm{Pr}_{\mathcal{M}}(s \models \Diamond T) = \mathrm{Pr}_{\mathcal{M}}(s' \models \Diamond T)$*

Two MCs $\mathcal{M}, \mathcal{M}'$ are bisimilar, denoted $\mathcal{M} \sim \mathcal{M}'$ if the initial states are bisimilar in the disjoint union of the MCs. It holds by definition that if $\mathcal{M} \sim \mathcal{M}'$, then $\mathrm{Pr}_{\mathcal{M}}(\Diamond T) = \mathrm{Pr}_{\mathcal{M}'}(\Diamond T')$. The notion of bisimulation can be lifted to pMCs [33].

[6] Alternatively, on acyclic models, a large step bound $h > |S|$ suffices.

**Idea 1:** Given a symbolic description $\mathcal{P}$ of a MC $[\![\mathcal{P}]\!]$, efficiently construct a concise MC $\mathcal{M}$ that is bisimilar to $\mathsf{CT}([\![\mathcal{P}]\!], h)$.

Indeed, the (compressed) CT in Fig. 4(b) and Fig. 4(a) are bisimilar. We remark that we do not necessarily compute the bisimulation quotient of $\mathsf{CT}([\![\mathcal{P}]\!], h)$.

## 4.2 Logical Perspective

The previous section defined weakly bisimilar chains and showed computational advantages, but did not present an algorithm. In this section we frame the finite horizon reachability probability as a logical query known as *weighted model counting* (WMC). In the next section we will show how this logical perspective yields an algorithm for constructing bisimilar MCs.

Weighted model counting is well-known as an effective reduction for probabilistic inference [16,57]. Let $\varphi$ be a logical sentence over variables $C$. The *weight function* $W_C : C \to \mathbb{R}_{\geq 0}$ assigns a weight to each logical variable. A *total variable assignment* $\eta : C \to \{0, 1\}$ by definition has weight $\mathsf{weight}(\eta) = \prod_{c \in C} W_C(c)\eta(c) + (1 - W_C(c)) \cdot (1 - \eta(c))$. Then the *weighted model count* for $\varphi$ given $W$ is $\mathsf{WMC}(\varphi, W_C) = \sum_{\eta \models \varphi} \mathsf{weight}(\eta)$. Formally, we desire to compute a reachability query using a WMC query in the following sense:

**Idea 2:** Given an MC $\mathcal{M}$, efficiently construct a predicate $\varphi^C_{\mathcal{M},h}$ and a weight-function $W_C$ such that $\Pr_{\mathcal{M}}(\Diamond^{\leq h} T) = \mathsf{WMC}(\varphi^C_{\mathcal{M},h}, W_C)$.

Consider initially the simplified case when the MC $\mathcal{M}$ is *binary*: every state has at most two successors. In this case producing $(\varphi^C_{\mathcal{M},h}, W_C)$ is straightforward:

*Example 5.* Consider the MC in Fig. 2(a), and note that it is binary. We introduce logical variables called *state/step coins* $C = \{c_{s,i} \mid s \in S, i < h\}$ for every state and step. Assignments to these coins denote choices of transitions at particular times: if the chain is in state $s$ at step $i$, then it takes the transition to the lexicographically first successor of $s$ if $c_{s,i}$ is true and otherwise takes the transition to the lexicographically second successor. To construct the predicate $\varphi^C_{\mathcal{M},3}$, we will need to write a logical sentence on coins whose models encode accepting paths (red paths) in the CT in Fig. 4(a).

We start in state $s = \langle 0, 0 \rangle$ (using state labels from the caption of Fig. 4). We order states as $s = \langle 0, 0 \rangle < t = \langle 0, 1 \rangle < u = \langle 1, 0 \rangle < v = \langle 1, 1 \rangle$. Then, $c_{s,0}$ is true if the chain transitions into state $s$ at time 0 and false if it transitions to state $t$ at time 0. So, one path from $s$ to the target node $\langle 1, 0 \rangle$ is given by the logical sentence $(c_{s,0} \wedge \neg c_{s,1} \wedge c_{t,2})$. The full predicate $\varphi^C_{\mathcal{M},3}$ is therefore:

$$\varphi^C_{\mathcal{M},3} = (c_{s,0} \wedge \neg c_{s,1} \wedge c_{t,2}) \vee (\neg c_{s,0} \wedge c_{t,1}) \vee (\neg c_{s,0} \wedge \neg c_{t,1} \wedge c_{v,2}).$$

Each model of this sentence is a single path to the target. This predicate $\varphi^C_{\mathcal{M},h}$ can clearly be constructed by considering all possible paths through the chain, but later on we will show how to build it more efficiently.

Finally, we fix $W_C$: The weight for each coin is directly given by the transition probability to the lexicographically first successor: for $0 \leq i < h$, $W_C(c_{s,i}) = 0.6$ and $W_C(c_{t,i}) = W_C(c_{v,i}) = 0.5$. The $\mathsf{WMC}$ is indeed 0.42, reflecting Example 1.

When the MC is not binary, it suffices to limit the out-degree of an MC to be at most two by adding auxiliary states, hence binarizing all transitions, cf. [38].

## 4.3 Connecting the Operational and the Logical Perspective

Now that we have reduced bounded reachability to weighted model counting, we reach a natural question: how do we perform $\mathsf{WMC}$?[7] Various approaches to performing $\mathsf{WMC}$ have been explored; a prominent approach is to compile the logical function into a binary decision diagram (BDD), which supports fast weighted model counting [21]. In this paper, we investigate the use of a BDD-driven approach for two reasons: (i) BDDs admit straightforward support for parametric models. (ii) BDDs provide a direct connection between the logical and operational perspectives. To start, observe that the graph of the BDD, together with the weights, can be interpreted as an MC:

**Definition 3.** *Let $\varphi^X$ be a propositional formula over variables $X$ and $<_X$ an ordering on $X$. Let $\mathsf{BDD}(\varphi^X, <_X) = \langle V, v_0, X, \mathsf{var}, \mathsf{val}, E_0, E_1\rangle$ be the corresponding BDD, and let $W$ be a weight function on $X$ with $0 \leq W(x) \leq 1$. We define the MC $\mathsf{BDD}_{\mathsf{MC}}(\varphi^X, <_X, W) = \langle S, \iota, P, T\rangle$ with $S = V$, $\iota = v_0$, $P(s) = \{E_0(s) \mapsto W(\mathsf{var}(s)), E_1(s) \mapsto 1 - W(\mathsf{var}(s))\}$ and $T = \{v \in V \mid \mathsf{val}(v) = 1\}$.*

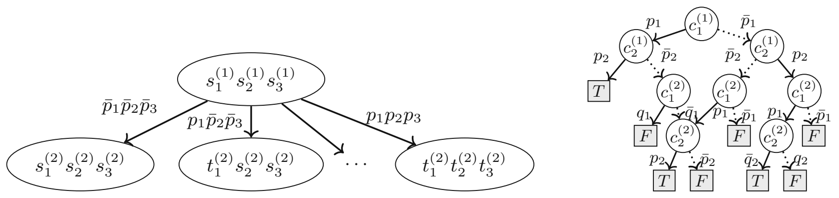
These BDDs are intimately related to the computation trees discussed before. For a binary MC $\mathcal{M}$, the tree $\mathsf{CT}(\mathcal{M}, h)$ is binary and can be considered as a (not necessarily reduced) BDD. More formally, let us construct $\mathsf{BDD}_{\mathsf{MC}}(\varphi^C_{\mathcal{M},h}, <_C,)$. We fix a total order on states. Then we fix *state/step coins* $C = \{c_{s,i} \mid s \in S, i < h\}$ and the weights as in Example 5. Finally, let $<_C$ be an order on $C$ such that $i < j$ implies $c_{s,i} <_C c_{s,j}$. Then:

$$\mathsf{CT}(\mathcal{M}, h) \sim \mathsf{BDD}_{\mathsf{MC}}(\varphi^C_{\mathcal{M},h}, <_C, W). \tag{2}$$

In the spirit of Idea 1, we thus aim to construct $\mathsf{BDD}_{\mathsf{MC}}(\varphi^C_{\mathcal{M},h}, <_C, W)$, a representation as outlined in Idea 2, efficiently. Indeed, the BDD (as MC) in Fig. 4(c) is bisimilar to the MC in Fig. 4(b).

**Idea 3:** Represent a bisimilar version of the computation tree using a BDD.

[7] In this paper, we concentrate on reductions to *exact* $\mathsf{WMC}$, leaving approximate approaches for future work [14].



(a) Unfactorized computation tree for ($h$=1, $n$=3). (b) Factorized ($h$=2, $n$=2).

**Fig. 5.** Two computation trees for the motivating example in Sect. 1.

## 4.4 The Algorithmic Benefits of BDD Construction

Thus far we have described how to construct a binarized MC bisimilar to the CT. Here, we argue that this construction has algorithmic benefits by filling in two details. First, the binarized representation is an important ingredient for compact BDDs. Second, we show how to choose a variable ordering that ensures that the BDDs grow linearly in the horizon. In sum,

**Idea 4:** WMC encodings of binarized Markov Chains may increase compression of computation trees.

To see the benefits of binarized transitions, we return to the factory example in Sect. 1. Figure 5(a) gives a bisimilar computation tree for the 3-factory $h = 1$ example. However, in this tree, the states are *unfactorized*: each node in the tree is a joint configuration of factories. This tree has 8 transitions (one for each possible joint state transition) with 8 distinct probabilities. On the other hand, the bisimilar computation tree in Fig. 1(d) has binarized transitions: each node corresponds to a single factory's state at a particular time-step, and each transition describes an update to only a single factory. This binarization enables the exploitation of new structure: in this case, the independence of the factories leads to smaller BDDs, that is otherwise lost when considering only joint configurations of factories.

Recall that the size of the ADD representation of the transition function is bounded from below by the number of distinct probabilities in the underlying MC: in this case, this is visualized by the number of distinct outgoing edge probabilities from all nodes in the unfactorized computation tree. Thus, a good binarization can have a drastically positive effect on performance. For the running example, rather than $2^n$ different transition probabilities (with $n$ factories), the system now has only $4 \cdot n$ distinct transition probabilities!

*Causal Orderings.* Next, we explore some of the *engineering choices* RUBICON makes to exploit the sequential structure in a MC when constructing the BDD for a WMC query. First, note that the transition matrix $P(s, s')$ implicitly encodes a distribution over state transition functions, $S \to S$. To encode $P$ as a BDD, we must encode each transition as a logical variable, similar to the situation in Sect. 4.2. In the case of binary transitions this is again easy. In the case of non-binary transitions, we again introduce additional logical variables [16,27,39,57].

This logical function has the following form:

$$f_P \colon \{0,1\}^C \to (S \to S). \tag{3}$$

Whereas the computation tree follows a fixed (temporal) order of states, BDDs can represent the same function (and the same weighted model count) using an arbitrary order. Note that the BDD's size and structure drastically depends both on the construction of the propositional formula *and* the order of the variables in that encoding. We can bound the size of the BDD by enforcing a variable order based on the temporal structure of the original MC. Specifically, given $h$ coin collections $\boldsymbol{C} = C \times \ldots \times C$, one can generate a function $f$ describing the $h$-length paths via repeated applications of $f_P$:

$$f \colon \{0,1\}^{\boldsymbol{C}} \to \text{Paths}_h \quad f(C_1, \ldots, C_h) = \Big(f_P(C_h) \circ \ldots \circ f_P(C_1)\Big)(\iota) \tag{4}$$

Let $\psi$ denote an indicator for the reachability property as a function over paths, $\psi \colon \text{Paths}_h \to \{0,1\}$ with $\psi(\pi) = [\pi \in [\![ \lozenge^{\leq h} T ]\!]]$. We call predicates formed by composition with $f_P$, i.e., $\varphi = \psi \circ f_P$, *causal encodings* and orderings on $c_{i,t} \in \boldsymbol{C}$ that are lexicographically sorted in time, $t_1 < t_2 \implies c_{i,t_1} < c_{j,t_2}$, *causal orderings.* Importantly, causally ordered / encoded BDDs grow linearly in horizon $h$, [61, Corollary 1]. More precisely, let $\varphi^{\boldsymbol{C}}_{\mathcal{M},h}$ be causally encoded where $|\boldsymbol{C}| = h \cdot m$. The causally ordered BDD for $\varphi^{\boldsymbol{C}}_{\mathcal{M},h}$ has at most $h \cdot |S \times S_\psi| \cdot m \cdot 2^m$ nodes, where $|S_\psi| = 2$ for reachability properties.[8] However, while the worst-case growth is linear in the horizon, constructing that BDD may induce a super-linear cost in the size, e.g., function composition using BDDs is super-linear!

Figure 5(b) shows the motivating factory example with 2 factories and $h = 2$. The variables are causally ordered: the factories in time step 1 occur before the factories in time step 2. For $n$ factories, a fixed number $f(n)$ of nodes are added to the BDD upon each iteration, guaranteeing growth on the order $\mathcal{O}(f(n) \cdot h)$. Note the factorization that occurs: the BDD has node sharing (node $c_2^{(2)}$ is reused) that yields additional computational benefits.

*Summary and Remaining Steps.* The operational view highlights that we want to compute a transformation of the original input MC $\mathcal{M}$. The logical view presents an approach to do so efficiently: By computing a BDD that stores a predicate describing all paths that reach the target, and interpreting and evaluating the (graph of the) BDD as an MC. In the following section, we discuss the two steps that we follow to create the BDD: (i) From $\mathcal{P}$ generate $\mathcal{P}'$ such that $\mathsf{CT}([\![ \mathcal{P} ]\!], h) \sim [\![ \mathcal{P}' ]\!]$. (ii) From $\mathcal{P}'$ generate $\mathcal{M}$ such that $\mathcal{M} = [\![ \mathcal{P}' ]\!]$.

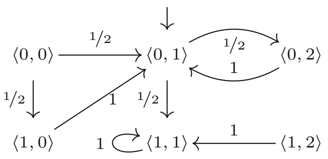
## 5 Rubicon

We present Rubicon which follows the two steps outlined above. For exposition, we first describe a translation of *monolithic* Prism programs to `Dice` programs

[8] Generally, it is the smallest number of states required for a DFA to recognize $\psi$.

```
module main
  x : [0..1] init 0;
  y : [0..2] init 1;
  [] x=0 & y<2 -> 0.5:x'=1 + 0.5:y'=y+1;
  [] y=2 -> 1:y'=y-1;
  [] x=1 & y!=2 -> 1:x'=y & y'=x;
endmodule
property: P=? [F<=2 (x=0 & y=2)]
```

(a) Prism program with reachability query



(b) Underlying MC

```
let s = init() in // init state
let T = hit(s) in  // init target
let (s, T) = if !T
   then let s' = step(s) in (s', hit(s'))
   else (s, T) in
let (s, T) = if !T then
   then let s' = step(s) in (s', hit(s'))
   else (s, T) in
T
```

(c) Main Dice program for $h$=2

```
fun init() { (0,1) }
fun hit((x,y)) { x ==0 && y == 2 }
fun step((x,y)) {
  if x==0 && y<2 then
     if flip 0.5 then (1,y) else (x,y+1)
  else if y==2 then (x,y-1)
  else if x==1 &&y!=1 then (y,x)
  else (x,y)
}
```

(d) Dice auxiliary functions

**Fig. 6.** From Prism to Dice using Rubicon.

and then extend this translation to admit modular programs. Technical steps and extensions are deferred to [38, Appendix].

**Dice Preliminaries.** We give a brief description of Dice, a probabilistic programming language (PPL) introduced in [39]. A PPL is a programming language augmented with a primitive notion of random choice: for instance, in Dice, a Bernoulli random variable is introduced by the syntax `flip 0.5`. The syntax of Dice is similar to the programming language OCaml: local variables are introduced by the syntax `let x = e`$_1$ `in e`$_2$, where `e`$_1$ and `e`$_2$ are *expressions*, i.e., sub-programs. Dice supports procedures, bounded integers, bounded loops, and standard control flow via `if`-statements.

One goal of a PPL is to perform *probabilistic inference*: compute the probability that the program returns a particular value. Inference on the tiny Dice program `let x = flip 0.1 in x` would yield that `true` is returned with probability 0.1. The Dice compiler performs probabilistic inference via weighted model counting and BDD compilation. In doing so, it accomplishes the *non-trivial* tasks of: (i) choosing a logical encoding for probabilistic programs (ii) establishing good variable orderings (iii) efficiently manipulating and constructing BDDs (iv) performing WMC . For details, we refer the reader to [39].

Rubicon uses Dice to effectively construct a BDD and perform WMC on a Dice program that reflects a description of some computation tree. This implementation exploits the structure that was described in Sect. 4.4: in particular, the BDD generated in Fig. 5(b) is exactly the BDD that will be generated by Dice from the output of Rubicon. The variable ordering used by Dice is given by the order in which program variables are introduced, and Rubicon's translation was designed with this variable ordering in mind.

**Transpiling Prism to Dice.** We present the core translation routine implemented in Rubicon. We note that the ultimate performance of Rubicon is heavily dependent on the quality of this translation. We evaluate the performance in the next section.

The PRISM specification language consists of one or more reactive *modules* (or partially synchronized state machines) that may interact with each other. Our example in Fig. 1(b) illustrates fully synchronized state machines. While PRISM programs containing multiple modules can be flattened into a single monolithic program, this yields an exponential blow-up: If one flattens the $n$ modules in Fig. 1(b) to a single module, the resulting program has $2^n$ updates per command. This motivates our direct translation of PRISM programs containing multiple modules.

*Monolithic Prism Programs.* We explain most ideas on PRISM programs that consist of a single "monolithic" module before we address the modular translation at the end of the subsection. A module has a set of bounded variables, and the valuations of these variables span the state space of the underlying MC. Its transitions are described by guarded *commands* of the form:

$$[\texttt{act}]\ \ \texttt{guard} \quad \rightarrow \quad p_1 : \texttt{update}_1 + \ldots\ldots + p_n : \texttt{update}_n$$

The *action* name `act` is only relevant in the modular case and can be ignored for now. The *guard* is a Boolean expression over the module's variables. If the guard evaluates to `true` for some state (a valuation), then the module evolves into one of the $n$ successor states by updating its variables. An *update* is chosen according to the probability distribution given by the expressions $p_1, \ldots, p_n$. In every state enabling the guard, the evaluation of $p_1, \ldots, p_n$ must sum up to one. A set of guards *overlap* if they all evaluate to `true` on a given state. The semantics of overlapping guards in the monolithic setting is to first uniformly select an active guard and then apply the corresponding stochastic transition. Finally, a self-loop is implicitly added to states without an enabled guard.

*Example 6.* We present our translation primarily through example. In Fig. 6(a), we give a PRISM program for a MC. The program contains two variables $x$ and $y$, where $x$ is either zero or one, and $y$ between zero and two. There are thus 6 different states. We denote states as tuples with the $x$- and $y$-value. We depict the MC in Fig. 6(b). From state $\langle 0, 0\rangle$, (only) the first guard is enabled and thus there are two transitions, each with probability a half: one in which $x$ becomes one and one in which $y$ is increased by one. Finally, there is no guard enabled in state $\langle 1, 1\rangle$, resulting in an implicit self-loop.

*Translation.* All `Dice` programs consist of two parts: a *main* routine, which is run by default when the program starts, and *function declarations* that declare auxiliary functions. We first define the auxiliary functions. For simplicity let us temporarily assume that no guards overlap and that probabilities are constants, i.e., not state-dependent.

The main idea in the translation is to construct a `Dice` function `step` that, given the current state, outputs the next state. Because a monolithic PRISM

```
module main
x : [0..2] init 1;
y : [0..2] init 1;
[] x>1 -> 1:x'=y&y'=x;
[] y<2 -> 1:x'=min(x+1,2);
endmodule
```

(a)

```
fun step((x,y)) {
  let aEn =(x>1)                      in
  let bEn =(y<2)                      in
  let act = selectFrom(aEn, bEn)    in
  if act==1 then (y,x)
  else if act==2 then (min(x+1,2),y)
  else (x,y)} ...
```

(b)

**Fig. 7.** PRISM program with overlapping guards and its translation (conceptually).

```
module m1
x : [0..1] init 0;
[a] x=1 -> 1:x'=1-y;
[b] x=0 -> 1:x'=0;
endmodule
module m2
y : [0..1] init 0;
[b] y=1 -> 0.5:y'=0 +0.5:y'=1;
[c] true -> 1:x'=1-x;
endmodule
```

(a)

```
fun step((x,y)) {
  let aEn =(x==1) in
  let bEn =(x=0 &&y=1) in
  let cEn =true in
  let act =selectFrom(aEn, bEn, cEn) in
  if act==1 then (1-y, y)
  else if act==2 then (0, flip 0.5)
  else if act==3 then (1-x, y)
  else (x,  y)
}
```

(b)

**Fig. 8.** Modular PRISM and resulting `Dice` step function.

program is almost a sequential program, in its most basic version, the `step` function is straightforward to construct using built-in `Dice` language primitives: we simply build a large if-else block corresponding to each command. This block iteratively considers each command's guard until it finds one that is satisfied. To perform the corresponding update we flip a coin – based on the probabilities corresponding to the updates – to determine which update to perform. If no command is enabled, we return the same state in accordance with the implicit self-loop. Figure 6(d) shows the program blocks for the PRISM program from Fig. 6(a) with target state $[\![x = 0, y = 2]\!]$. There are two other important auxiliary functions. The `init` function simply returns the initial state by translating the initialization statements from PRISM, and the `hit` function checks whether the current state is a target state that is obtained from the property.

Now we outline the main routine, given for this example in Fig. 6(c). This function first initializes the state. Then, it calls `step` 2 times, checking on each iteration using `hit` if the target state is reached. Finally, we return whether we have been in a target state. The probability to return true corresponds to the reachability probability on the underlying MC specified by the PRISM program.

*Overlapping Guards.* PRISM allows multiple commands to be enabled in the same state, with semantics to uniformly at random choose one of the enabled commands to evaluate. `Dice` has no primitive notion of this construct.[9] We illustrate the translation in Fig. 7(a) and Fig. 7(b). It determines which guards `aEn`, `bEn`, `cEn` are enabled. Then, we randomly select one of the commands which are enabled, i.e., we uniformly at random select a true bit from a given tuple

[9] One cannot simply condition on selecting an enabled guard as this redistributes probability mass over all paths and not only over paths with the same prefix.

of bits. We store the index of that bit and use it to execute the corresponding command.

*Modular Prism Programs.* For modular PRISM programs, the *action names* at the front of PRISM commands are important. In each module, there is a set of action names available. An action is *enabled* if each module that contains this action name has (at least) one command with this action whose guard is satisfied. Commands with an empty action are assumed to have a globally unique action name, so in that case the action is enabled iff the guard is enabled. Intuitively, once an action is selected, we randomly select a command per module in all modules containing this action name. Our approach resembles that for overlapping guards described above. See Fig. 8 for an intuitive example. To automate this, the updates require more care, cf. [38] for details.

*Implementation.* RUBICON is implemented on top of STORM's Python API and translates PRISM to `Dice` fully automatically. RUBICON supports all MCs in the PRISM benchmark suite and a large set of benchmarks from the PRISM website and the QVBS [35], with the note that we require a single initial state and ignore reward declarations. Furthermore, we currently do not support the hide/restrict process-algebraic compositions and some integer operations.

## 6 Empirical Comparisons

We compare and contrast the performance of STORM against RUBICON to empirically demonstrate the following strengths and weaknesses:[10]

**Explicit Model Checking (STORM)** represents the MC explicitly in a sparse matrix format. The approach suffers from the state space explosion, but has been engineered to scale to models with many states. Besides the state space, the sparseness of the transition matrix is essential for performance.

**Symbolic Model Checking (STORM)** represents the transition matrix and the reachability probability as an ADD. This method is strongest when the transition matrix and state vector have structure that enables a small ADD representation, like symmetry and sparsity.

**RUBICON** represents the set of paths through the MC as a (logical) BDD. This method excels when the state space has structure that enables a compact BDD representation, such as conditional independence, and hence scales well on examples with many (asymmetric) parallel processes or queries that admit a compact representation.

The sources, benchmarks and binaries are archived.[11]

There is no clear-cut model checking technique that is superior to others (see QCOMP [12]). We demonstrate that, while RUBICON is not competitive on some

[10] All experiments were conducted with STORM version 1.6.0 on the same server with 512 GB of RAM, using a single thread of execution. Time was reported using the built-in Unix `time` utility; the total wall-clock time is reported.

[11] http://doi.org/10.5281/zenodo.4726264 and http://github.com/sjunges/rubicon.

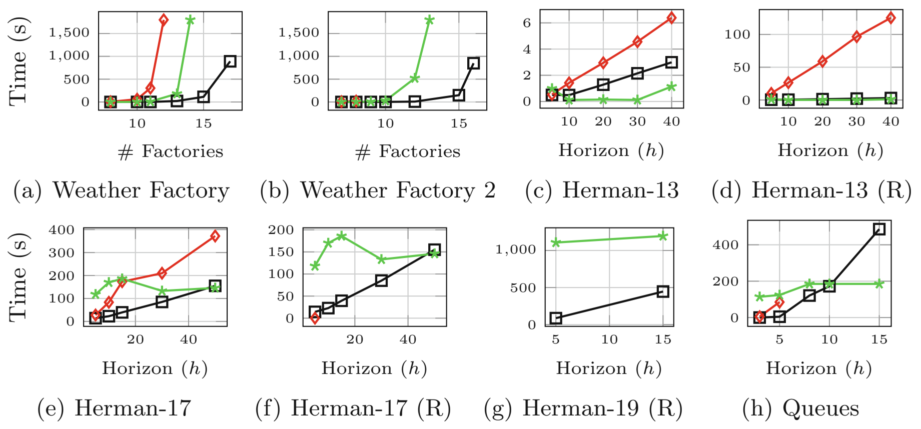


**Fig. 9.** Scaling plots comparing Rubicon (—□—), Storm's symbolic engine (—◇—), and Storm's explicit engine (—*—). An "(R)" in the caption denotes random parameters.

commonly used benchmarks [52], it improves a modern model checking portfolio approach on a significant set of benchmarks. Below we provide several natural models on which Rubicon is superior to one or both competing methods. We also evaluated Rubicon on standard benchmarks, highlighting that Rubicon is applicable to models from the literature. We see that Rubicon is effective on Herman (elaborated below), has mixed results on BRP [38, Appendix], and is currently not competitive on some other standard benchmarks (NAND, EGL, LeaderSync). While not exhaustive, our selected benchmarks highlight specific strengths and weaknesses of Rubicon. Finally, a particular benefit of Rubicon is fast sampling of parametric chains, which we demonstrate on Herman and our factory example.

**Scaling Experiments.** In this section, we describe several scaling experiments (Fig. 9), each designed to highlight a specific strength or weakness.

*Weather Factories.* First, Fig. 9(a) describes a generalization of the motivating example from Sect. 1. In this model, the probability that each factory is on strike is dependent on a common random event: whether or not it is raining. The rain on each day is dependent on the previous day's weather. We plot runtime for an increasing number of factories for $h$=10. Both Storm engines eventually fail due to the state explosion and the number of distinct probabilities in the MC. Rubicon is orders of magnitude faster in comparison, highlighting that it does not depend on complete independence among the factories. Figure 9(b) shows a more challenging instance where the weather includes *wind* which, each day, affects whether or not the sun will shine, which in turn affects strike probability.

*Herman.* Herman is based on a distributed protocol [37] that has been well-studied [1,53] and which is one of the standard benchmarks in probabilistic model checking. Rather than computing the expected steps to 'stabilization', we consider the step-bounded probability of stabilization. Usually, all participants in the protocol flip a coin with the same bias. The model is then highly symmetric, and hence is amenable to symbolic representation with ADDs. Figures 9(c) and 9(e) show how the methods scale on Herman examples with 13 and 17 parallel processes. We observe that the explicit approach scales very efficiently in the number of iterations but has a much higher up-front model-construction cost, and hence can be slower for fewer iterations.

To study what happens when the coin biases vary over the protocol participants, we made a version of the Herman protocol where each participant's bias is randomly chosen, which ruins the symmetry and so causes the ADD-based approaches to scale significantly worse (Figs. 9(d) and 9(f), and 9(g)); we see that symbolic ADD-based approaches completely fail on Herman 17 and Herman 19 (the curve terminating denotes a memory error). Rubicon and the explicit approach are unaffected by varying parameters.

*Queues.* The Queues model has $K$ queues of capacity $Q$ where every step, tasks arrive with a particular probability. Three queues are of type 1, the others of type 2. We ask the probability that all queues of type 1 and at least one queue of type 2 is full within $k$ steps. Contrary to the previous models, the ADD representation of the transition matrix is small. Figure 9(h) shows the relative scaling on this model with $K = 8$ and $Q = 3$. We observe that ADDs quickly fail due to inability to concisely represent the probability vector $\boldsymbol{x}$ from Sect. 3. Rubicon outperforms explicit model checking until $h = 10$.

**Sampling Parametric Markov Chains.** We evaluate performance for the pMC sampling problem outlined in Sect. 2. Table 1 gives for four models the time to construct the BDD and to perform WMC, as well as the time to construct an ADD in Storm and to perform model checking with this ADD. Finally, we show the time for Storm to compute the solution function of the pMC (with the explicit representation). The pMC sampling in Storm – symbolic and explicit – computes the reachability probabilities with concrete probabilities. Rubicon, in contrast, constructs a 'parametric' BDD once, amortizing the cost of repeated efficient evaluation. The 'parametric BDD' may be thought of as a solution function, as discussed in Sect. 4.1. Storm cannot compute these solution functions as efficiently. We observe in Table 1 that fast parametric sampling is realized in Rubicon: for instance, after a 40s up-front compilation of the factories example with 15 factories, we have a solution function in factorized form and it costs an order of magnitude less time to draw a sample. Hence, sampling and computation of solution functions of pMCs is a major strength of Rubicon.

## 7 Discussion, Related Work, and Conclusion

We have demonstrated that the probabilistic inference approach to probabilistic model checking can improve scalability on an important class of problems. Another benefit of the approach is for sampling pMCs. These are used to evaluate e.g., robustness of systems [1], or to synthesise POMDP controllers [41]. Many state-of-the-art approaches [17,19,24] require the evaluation of various instantiated MCs, and Rubicon is well-suited to this setting. More generally, support

**Table 1.** Sampling performance comparison and pMC model checking, time in seconds.

| Model | RUBICON | | STORM (w/ ADD) | | STORM (explicit) |
|---|---|---|---|---|---|
| | Build | WMC | Build | Solve | pMC solving |
| Herman R 13 ($h = 10$) | 3 | <1 | 32 | 18 | >1800 |
| Herman R 17 ($h = 10$) | 45 | 28 | >1800 | – | >1800 |
| Factories 12 ($h = 15$) | 2 | <1 | 59 | 286 | >1800 |
| Factories 15 ($h = 15$) | 40 | 4 | >1800 | – | >1800 |

of inference techniques opens the door to a variety of algorithms for additional queries, e.g., computing *conditional probabilities* [3,8].

An important limitation of probabilistic inference is that only finitely many paths can be stored. For infinite horizon properties in cyclic models, an infinite set of arbitrarily long paths would be required. However, as standard in probabilistic model checking, we may soundly approximate infinite horizons. Additionally, the inference algorithm in `Dice` does not support a notion of non-determinism. It thus can only be used to evaluate MCs, not Markov decision processes. However, [61] illustrates that this is not a conceptual limitation. Finally, we remark that RUBICON achieves its performance with a straightforward translation. We are optimistic that this is a first step towards supporting a larger class of models by improving the transpilation process for specific problems.

**Related Work.** The tight connection with inference has been recently investigated via the use of model checking for Bayesian networks, the prime model in probabilistic inference [56]. Bayesian networks can be described as probabilistic programs [10] and their operational semantics coincides with MCs [31]. Our work complements these insights by studying how symbolic model checking can be sped up by probabilistic inference.

The path-based perspective is tightly connected to *factored state spaces*. Factored state spaces are often represented as (bipartite) Dynamic Bayesian networks. ADD-based model checking for DBNs has been investigated in [25], with mixed results. Their investigation focuses on using ADDs for factored state space representations. We investigate using BDDs representing paths. Other approaches also investigated a path-based view: The symbolic encoding in [28] annotates propositional sub-formulae with probabilities, an idea closer to ours. The underlying process implicitly constructs an (uncompressed) CT leading to an exponential blow-up. Likewise, an explicit construction of a computation tree without factorization is considered in [62]. Compression by grouping paths has been investigated in two *approximate* approaches: [55] discretises probabilities and encodes into a satisfiability problem with quantifiers and bit-vectors. This idea has been extended [60] to a PAC algorithm by purely propositional encodings and (approximate) model counting [14]. Finally, factorisation exploits symmetries, which can be exploited using symmetry reduction [50]. We highlight that the latter is not applicable to the example in Fig. 1(d).

There are many techniques for exact probabilistic inference in various forms of probabilistic modeling, including probabilistic graphical models [20,54]. The semantics of graphical models make it difficult to transpile PRISM programs, since commonly used operations are lacking. Recently, *probabilistic programming languages* have been developed which are more amenable to transpilation [13,23,29,30,59]. We target `Dice` due to the technical development that it enables in Sect. 4, which enabled us to design and explain our experiments. Closest related to `Dice` is ProbLog [27], which is also a PPL that performs inference via WMC; ProbLog has different semantics from `Dice` that make the translation less straightforward. The paper [61] uses an encoding similar to `Dice` for inferring specifications based on observed traces. ADDs and variants have been considered for probabilistic inference [15,18,58], which is similar to the process commonly used for probabilistic model checking. The planning community has developed their own disjoint sets of methods [45]. Some ideas from learning have been applied in a model checking context [11].

## 8 Conclusion

We present RUBICON, bringing probabilistic AI to the probabilistic model checking community. Our results show that RUBICON can outperform probabilistic model checkers on some interesting examples, and that this is not a coincidence but rather the result of a significantly different perspective.

## References

1. Aflaki, S., Volk, M., Bonakdarpour, B., Katoen, J.P., Storjohann, A.: Automated fine tuning of probabilistic self-stabilizing algorithms. In: SRDS, pp. 94–103. IEEE (2017)
2. de Alfaro, L., Kwiatkowska, M., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 395–410. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_27
3. Andrés, M.E., van Rossum, P.: Conditional probabilities over probabilistic and nondeterministic systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 157–172. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_12
4. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
5. Baier, C., Hensel, C., Hutschenreiter, L., Junges, S., Katoen, J.P., Klein, J.: Parametric Markov chains: PCTL complexity and fraction-free Gaussian elimination. Inf. Comput. **272**, 104504 (2020)
6. Baier, C., Hermanns, H.: Weak bisimulation for fully probabilistic processes. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 119–130. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_14

7. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
8. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 515–530. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_43
9. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: CCS, pp. 1249–1264. ACM (2019)
10. Batz, K., Kaminski, B.L., Katoen, J.-P., Matheja, C.: How long, O Bayesian network, will I sample thee? - a program analysis perspective on expected sampling times. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 186–213. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_7
11. Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
12. Budde, C.E., et al.: On correctness, precision, and performance in quantitative verification: QComp 2020 competition report. In: ISOLA. LNCS. Springer, Heidelberg (2020)
13. Carpenter, B., et al.: Stan: a probabilistic programming language. J. Stat. Soft. **VV**(Ii) (2016)
14. Chakraborty, S., Fried, D., Meel, K.S., Vardi, M.Y.: From weighted to unweighted model counting. In: IJCAI, pp. 689–695. AAAI Press (2015)
15. Chavira, M., Darwiche, A.: Compiling Bayesian networks using variable elimination. In: IJCAI, pp. 2443–2449 (2007)
16. Chavira, M., Darwiche, A.: On probabilistic inference by weighted model counting. Artif. Intell. **172**(6–7), 772–799 (2008)
17. Chen, T., Hahn, E.M., Han, T., Kwiatkowska, M.Z., Qu, H., Zhang, L.: Model repair for Markov decision processes. In: TASE, pp. 85–92. IEEE (2013)
18. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: FSE, pp. 92–102 (2013)
19. Cubuktepe, M., Jansen, N., Junges, S., Katoen, J.P., Topcu, U.: Scenario-based verification of uncertain MDPs. In: TACAS. LNCS, vol. 12078, pp. 287–305. Springer, Heidelberg (2020)
20. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: IJCAI, pp. 819–826 (2011)
21. Darwiche, A., Marquis, P.: A knowledge compilation map. JAIR **17**, 229–264 (2002)
22. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31862-0_21
23. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: a probabilistic prolog and its application in link discovery. In: IJCAI, vol. 7, pp. 2462–2467 (2007)
24. Dehnert, C., et al.: `PROPhESY`: a PRObabilistic ParamEter SYnthesis tool. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 214–231. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_13
25. Deininger, D., Dimitrova, R., Majumdar, R.: Symbolic model checking for factored probabilistic models. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 444–460. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_28
26. van Dijk, T., van de Pol, J.: Multi-core symbolic bisimulation minimisation. STTT **20**(2), 157–177 (2018)

27. Fierens, D., et al.: Inference and learning in probabilistic logic programs using weighted Boolean formulas. Theory Pract. Log. Prog. **15**(3), 358–401 (2015)
28. Fränzle, M., Hermanns, H., Teige, T.: Stochastic satisfiability modulo theory: a novel technique for the analysis of probabilistic hybrid systems. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 172–186. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78929-1_13
29. Gehr, T., Misailovic, S., Vechev, M.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 62–83. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_4
30. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
31. Gretz, F., Katoen, J.P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Eval. **73**, 110–132 (2014)
32. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: abstraction refinement for infinite probabilistic models. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_30
33. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. STTT **13**(1), 3–19 (2011)
34. Hartmanns, A., Hermanns, H.: The modest toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
35. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS. LNCS, vol. 11427, pp. 344–350. Springer, Heidelberg (2019)
36. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker storm. STTT (2021, to appear)
37. Herman, T.: Probabilistic self-stabilization. Inf. Process. Lett. **35**(2), 63–67 (1990)
38. Holtzen, S., Junges, S., Vazquez-Chanlatte, M., Millstein, T., Seshia, S.A., Van den Broeck, G.: Model checking finite-horizon Markov chains with probabilistic inference. CoRR abs/2105.12326 (2021)
39. Holtzen, S., Van den Broeck, G., Millstein, T.: Scaling exact inference for discrete probabilistic programs. PACMPL OOPSLA, November 2020
40. Jansen, D.N., Groote, J.F., Timmers, F., Yang, P.: A near-linear-time algorithm for weak bisimilarity on Markov chains. In: CONCUR. LIPIcs, vol. 171, pp. 8:1–8:20. Schloss Dagstuhl - LZI (2020)
41. Junges, S., et al.: Finite-state controllers of POMDPs using parameter synthesis. In: UAI, pp. 519–529. AUAI Press (2018)
42. Katoen, J.-P., Gretz, F., Jansen, N., Kaminski, B.L., Olmedo, F.: Understanding probabilistic programs. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Correct System Design. LNCS, vol. 9360, pp. 15–32. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23506-6_4
43. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation minimisation mostly speeds up probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_9
44. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. FMSD **36**(3), 246–280 (2010)

45. Klauck, M., Steinmetz, M., Hoffmann, J., Hermanns, H.: Bridging the gap between probabilistic model checking and probabilistic planning: survey, compilations, and empirical comparison. JAIR **68**, 247–310 (2020)
46. Klein, J., et al.: Advances in probabilistic model checking with PRISM: variable reordering, quantiles and weak deterministic büchi automata. STTT **20**(2), 179–194 (2018)
47. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press, Cambridge (2009)
48. Kozen, D.: Semantics of probabilistic programs. JCSS **22**(3), 328–350 (1981)
49. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: a hybrid approach. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 52–66. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_5
50. Kwiatkowska, M., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 234–248. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_23
51. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
52. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: QEST, pp. 203–204. IEEE (2012)
53. Kwiatkowska, M.Z., Norman, G., Parker, D.: Probabilistic verification of Herman's self-stabilisation algorithm. Formal Aspects Comput. **24**(4–6), 661–670 (2012)
54. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1988)
55. Rabe, M.N., Wintersteiger, C.M., Kugler, H., Yordanov, B., Hamadi, Y.: Symbolic approximation of the bounded reachability probability in large Markov chains. In: Norman, G., Sanders, W. (eds.) QEST 2014. LNCS, vol. 8657, pp. 388–403. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10696-0_30
56. Salmani, B., Katoen, J.-P.: Bayesian inference by symbolic model checking. In: Gribaudo, M., Jansen, D.N., Remke, A. (eds.) QEST 2020. LNCS, vol. 12289, pp. 115–133. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59854-9_9
57. Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI, vol. 5, pp. 475–481 (2005)
58. Smolka, S., et al.: Scalable verification of probabilistic networks. In: PLDI, pp. 190–203. ACM (2019)
59. van de Meent, J.W., Paige, B., Yang, H., Wood, F.: An Introduction to Probabilistic Programming. arXiv:1809.10756 (2018)
60. Vazquez-Chanlatte, M., Rabe, M.N., Seshia, S.A.: A model counter's guide to probabilistic systems. CoRR abs/1903.09354 (2019)
61. Vazquez-Chanlatte, M., Seshia, S.A.: Maximum causal entropy specification inference from demonstrations. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 255–278. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_15
62. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 366–380. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-93900-9_29

# Enforcing Almost-Sure Reachability in POMDPs

Sebastian Junges[1](✉), Nils Jansen[2], and Sanjit A. Seshia[1]

[1] University of California at Berkeley, Berkeley, USA
sjunges@berkeley.edu

[2] Radboud University Nijmegen, Nijmegen, The Netherlands

**Abstract.** Partially-Observable Markov Decision Processes (POMDPs) are a well-known stochastic model for sequential decision making under limited information. We consider the EXPTIME-hard problem of synthesising policies that almost-surely reach some goal state without ever visiting a bad state. In particular, we are interested in computing the winning region, that is, the set of system configurations from which a policy exists that satisfies the reachability specification. A direct application of such a winning region is the safe exploration of POMDPs by, for instance, restricting the behavior of a reinforcement learning agent to the region. We present two algorithms: A novel SAT-based iterative approach and a decision-diagram based alternative. The empirical evaluation demonstrates the feasibility and efficacy of the approaches.

## 1 Introduction

Partially observable Markov decision processes (POMDPs) constitute the standard model for agents acting under partial information in uncertain environments [34,52]. A common problem is to find a policy for the agent that maximizes a reward objective [36]. This problem is undecidable, yet, well-established approximate [27], point-based [43], or Monte-Carlo-based [49] methods exist. In safety-critical domains, however, one seeks a *safe* policy that exhibits strict behavioral guarantees, for instance in the form of temporal logic constraints [44]. The aforementioned methods are not suitable to deliver provably safe policies. In contrast, we employ almost-sure reach-avoid specifications, where the probability to reach a set of *avoid* states is zero, and the probability to *reach* a set of goal states is one. Our **Challenge 1** is to compute a policy that adheres to such specifications. Furthermore, we aim to ensure the *safe exploration of a POMDP*, with safe reinforcement learning [23] as direct application. **Challenge 2** is then

This work is partially supported by NSF grants 1545126 (VeHICaL), 1646208 and 1837132, by the DARPA contracts FA8750-18-C-0101 (AA) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, and by Toyota under the iCyPhy center.
This research has been partially funded by NWO grant OCENW.KLEIN.187: "Provably Correct Policies for Uncertain Partially Observable Markov Decision Processes".

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 602–625, 2021.
https://doi.org/10.1007/978-3-030-81688-9_28

to compute a large set of safe policies for the agent to choose from at any state of the POMDP. Such sets of policies are called *permissive policies* [21,31].

*POMDP Almost-Sure Reachability Verification.* Let us remark that in POMDPs, we cannot directly observe in which state we are, but we are in general able to track a *belief*, i.e., a distribution over states that describes where in the POMDP we may be. The belief allows us to formulate the following **verification task**:

For a POMDP, sets of target and avoid states, and a belief, does a policy exist such that we reach the target states without ever visiting a bad state?

The underlying EXPTIME-complete problem requires—in general—policies with access to memory of exponential size in the number of states [4,18]. For safe exploration and, e.g., to support nested temporal properties, the ability to solve this problem *for each belief in the POMDP* is essential.

We base our approaches on the concept of a *winning region*, also referred to as controllable or attractor regions. Such regions are sets of *winning beliefs* from which a policy exists that guarantees to satisfy an almost-sure specification. The verification task relates three concrete problems which we tackle in this paper: (1) *Decide* whether a belief is winning, (2) *compute* the *maximal* winning region, and (3) *compute* a *large* yet not necessarily maximal winning region. We now outline our two approaches. First, we directly exploit model checking for MDPs [5] using belief abstractions. The second, much faster approach iteratively exploits *satisfiability solving* (SAT) [8]. Finally, we define a scheme to enable safe reinforcement learning [23] for POMDPs, referred to as *shielding* [2,30].

*MDP Model Checking.* A prominent approach gives the semantics of a POMDP via an (infinite) belief MDP whose states are the beliefs in the POMDP [36]. For almost-sure specifications, it is sufficient to consider *belief-supports* rather than beliefs. In particular, two beliefs with the same support are either both in a winning region or not [47]. We abstract a belief MDP into a finite belief-support MDP, whose states are the support of beliefs. The (maximal) winning region are (all) states of the belief-support MDP from which one can almost surely reach a belief support that contains a goal state without visiting belief support states that contain an avoid state.

To find a winning region in the POMDP, we thus just have to solve almost-sure reachability in this finite MDP. The number of belief supports, however, is exponentially large in the number of POMDP states, threatening the efficient application of explicit state verification approaches. Symbolic state space representations are a natural option to mitigate this problem [7]. We construct a symbolic description of the belief support MDP and apply state-of-the-art symbolic model checking. Our experiments show that this approach (referred to as *MDP Model Checking*) does in general not alleviate the exponential blow-up.

*Incremental SAT Solving.* While the belief support model exploits the structure of the belief support MDP by using a symbolic state space representation, it does not exploit elementary properties of the structure of winning regions. To overcome the scalability challenge, we aim to exploit information from the original POMDP, rather than working purely on the belief-support MDP. In a nutshell, our approach computes the winning regions in a backward fashion by *optimistically* searching policies without memory on the POMDP level. Concretely, starting from the belief support states that shall be reached almost-surely, further states are added to the winning region if we quickly can find a policy that reaches these states without visiting those that are to avoid. We search for these policies by incrementaly employing an encoding based on SAT solving. This symbolic encoding avoids an expensive construction of the belief support MDP. The computed winning region directly translates to sufficient constraints on the set of safe policies, i.e., each policy satisfying these constraints satisfies, by construction, the specification. The key idea is to successively add short-cuts corresponding to already known safe policies. These changes to the structure of the POMDP are performed implicitly on the SAT encoding. The resulting scalable method is sound, but not complete by itself. However, it can be rendered complete by trading off a certain portion of the scalability; intuitively one would eventually search for policies with larger amounts of memory.

*Shielding.* An agent that stays within a winning region is guaranteed to adhere to the specification. In particular, we *shield* (or *mask*) any action of the agent that may lead out of the winning region [1,39,42]. We stress that the shape of the winning region is independent of the transition probabilities or rewards in the POMDP. This independence means that the only prior knowledge we need to assume is the topology, that is, the graph of the POMDP. A pre-computation of the winning region thus yields a shield and allows us to restrict an agent to safely explore environments, which is the essential requirement for safe reinforcement learning [22,23] of POMDPs. The shield can be used with any RL agent [2].

*Comparison with the State-of-the-Art.* Similar to our approach, [15] solves almost-sure specifications using SAT. Intuitively, the aim is to find a so-called *simple policy* that is Markovian (aka memoryless). Such a policy may not exist, yet, the method can be applied to a POMDP that has an extended state space to account for finite memory [33,37]. There are three shortcomings that our incremental SAT approach overcomes. First, one needs to pre-define the memory a policy has at its disposal, as well as a fixed lookahead on the exploration of the POMDP. Our encoding does not require to fix these hyperparameter a priori. Second, the approach is only feasible if small memory bounds suffice. Our approach scales to models that require policies with larger memory bounds. Third, the approach finds a single simple policy starting from a pre-defined initial state. Instead, we find a large winning region. For safe exploration, this means that we may exclude many policies and never explore important parts of the system, harming the final performance of the agent. Shielding MDPs is not new [2,9,10,30]. However, those methods do neither take partial observability into account, nor can they guarantee reaching desirable states. Nam and Alur [39] cover partial observability and reachability, but do not account for stochastic uncertainty.

*Experiments.* To showcase the feasibility of our method, we adopted a number of typical POMDP environments. We demonstrate that our method scales better than the state of the art. We evaluate the shield by letting an agent explore the

POMDP environment according to the permissive policy, thereby enforcing the satisfaction of the almost-sure specification. We visualize the resulting behavior of the agent in those environments with a set of videos.

*Contributions.* Our paper makes four contributions: (1) We present an incremental SAT-based approach to compute policies that satisfy almost-sure properties. The method scales to POMDPs whose belief-support states count billions; (2) The novel approach is able to find large winning regions that yield permissive policies. (3) We implement a straightforward approach that constructs the belief-support symbolically using state-of-the-art model checking. We show that its completeness comes at the cost of limited scalability. (4) We construct a shield for almost-sure specifications on POMDPs which enforces at runtime that *no unsafe states are visited* and that, under mild assumptions, *the agent almost-surely reaches the set of desirable states*.

*Further Related Work.* Chatterjee et al. compute winning regions for minimizing a reward objective via an explicit state representation [17], or consider almost-sure reachability using an explicit state space [16,51]. The problem of determining any winning policy can be cast as a strong cyclic planning problem, proposed earlier with decision diagrams [7]. Indeed, our BDD-based implementation on the belief-support MDP can be seen as a reimplementation of that approach.

Quantitative variants of reach-avoid specifications have gained attention in, e.g., [11,28,40]. Other approaches restrict themselves to simple policies [3,33,45,58]. Wang et al. [55] use an iterative Satisfiability Modulo Theories (SMT) [6] approach for quantitative finite-horizon specifications, which requires computing beliefs. Various general POMDP approaches exist, e.g., [26,27,29,48,49,54,56]. The underlying approaches depend on discounted reward maximization and can satisfy almost-sure specifications with high reliability. However, enforcing probabilities that are close to 0 or 1 requires a discount factor close to 1, drastically reducing the scalability of such approaches [28]. Moreover, probabilities in the underlying POMDP need to be precisely given, which is not always realistic [14].

Another line of work (for example [53]) uses an idea similar to winning regions with uncertain specifications, but in a fully observable setting. Finally, complementary to shielding, there are approaches that guide reinforcement learning (with full observability) via temporal logic constraints [24,25].

## 2 Preliminaries and Formal Problem

We briefly introduce POMDPs and their semantics in terms of belief MDPs, before formalising and studying the problem variants outlined in the introduction. We present belief-support MDPs as a finite abstraction of infinite belief MDPs.

We define the support $supp(\mu) = \{x \in X \mid \mu(x) > 0\}$ of a discrete probability distribution $\mu$ and denote the set of all distributions with $Distr(X)$.

**Definition 1 (MDP).** *A* Markov decision process *(MDP) is a tuple* $\mathcal{M} = \langle S, \mathrm{Act}, \mu_{\mathrm{init}}, \mathbf{P} \rangle$ *with a set* $S$ *of states, an initial distribution* $\mu_{\mathrm{init}} \in Distr(S)$*, a finite set* Act *of actions, and a transition function* $\mathbf{P} \colon S \times \mathrm{Act} \to Distr(S)$*.*

Let $\text{post}_s(\alpha) = \mathit{supp}(\mathbf{P}(s,\alpha))$ denote the states that may be the successors of the state $s \in S$ for action $\alpha \in \text{Act}$ under the distribution $\mathbf{P}(s,\alpha)$. If $\text{post}_s(\alpha) = \{s\}$ for all actions $\alpha$, $s$ is called *absorbing*.

**Definition 2 (POMDP).** *A* partially observable MDP *(POMDP) is a tuple* $\mathcal{P} = \langle \mathcal{M}, \Omega, \text{obs} \rangle$ *with* $\mathcal{M} = \langle S, \text{Act}, \mu_{\text{init}}, \mathbf{P} \rangle$ *the underlying MDP with finite* $S$*,* $\Omega$ *a finite set of observations, and* $\text{obs}\colon S \to \Omega$ *an observation function. We assume that there is a unique initial observation, i.e., that* $|\{\text{obs}(s) \mid s \in \mathit{supp}(\mu_{\text{init}})\}| = 1$.

More general observation functions $\text{obs}\colon S \to \mathit{Distr}(\Omega)$ are possible via a (polynomial) reduction [17]. A path through an MDP is a sequence $\pi$, $\pi = (s_0, \alpha_0)(s_1, \alpha_1) \dots s_n$ of states and actions. such that $s_{i+1} \in \text{post}_{s_i}(\alpha_i)$ for $\alpha_i \in \text{Act}$ and $0 \leq i < n$. The observation function obs applied to a path yields an observation(-action) sequence $\text{obs}(\pi)$ of observations and actions.

For modeling flexibility, we allow actions to be unavailable in a state (e.g., opening doors is only available when at a door), and it turned out to be crucial to handle this explicitly in the following algorithms. Technically, the transition function is a partial function, and the enabled actions are a set $\text{EnAct}(s) = \{\alpha \in \text{Act} \mid \text{post}_s(\alpha) \neq \emptyset\}$. To ease the presentation, we assume that states $s, s'$ with the same observation share a set of enabled actions $\text{EnAct}(s) = \text{EnAct}(s')$.

**Definition 3 (Policy).** *A policy* $\sigma\colon (S \times \text{Act})^* \times S \to \mathit{Distr}(\text{Act})$ *maps a path* $\pi$ *to a distribution over actions. A policy is* observation-based*, if for each two paths* $\pi$*,* $\pi'$ *it holds that* $\text{obs}(\pi) = \text{obs}(\pi') \Rightarrow \sigma(\pi) = \sigma(\pi')$*. A policy is* memoryless*, if for each* $\pi$*,* $\pi'$ *it holds that* $\mathsf{last}(\pi) = \mathsf{last}(\pi') \Rightarrow \sigma(\pi) = \sigma(\pi')$*. A policy is* deterministic*, if for each* $\pi$*,* $\sigma(\pi)$ *is a Dirac distribution, i.e., if* $|\mathit{supp}(\sigma(\pi))| = 1$*.*

Policies resolve nondeterminism and partial observability by turning a (PO)MDP into the *induced* infinite discrete-time Markov chain whose states are the finite paths of the (PO)MDP. Probability measures are defined on this Markov chain.

For POMDPs, a *belief* describes the probability of being in certain state based on an observation sequence. Formally, a belief $\mathfrak{b}$ is a distribution $\mathfrak{b} \in \mathit{Distr}(S)$ over the states. A state $s$ with positive belief $\mathfrak{b}(s) > 0$ is in the *belief support*, $s \in \mathit{supp}(b)$. Let $\mathit{Pr}^{\sigma}_{\mathfrak{b}}(S')$ denote the probability to reach a set $S' \subseteq S$ of states from belief $\mathfrak{b}$ under the policy $\sigma$. More precisely, $\mathit{Pr}^{\sigma}_{\mathfrak{b}}(S')$ denotes the probability of all paths that reach $S'$ from $\mathfrak{b}$ when nondeterminism is resolved by $\sigma$.

The policy synthesis problem usually consists in finding a policy that satisfies a certain specification for a POMDP. We consider *reach-avoid* specifications, a subclass of indefinite horizon properties [46]. For a POMDP $\mathcal{P}$ with states $S$, such a specification is $\varphi = \langle \mathit{REACH}, \mathit{AVOID} \rangle \subseteq S \times S$. We assume that states in $\mathit{AVOID}$ and in $\mathit{REACH}$ are (made) absorbing and $\mathit{REACH} \cap \mathit{AVOID} = \emptyset$.

**Definition 4 (Winning).** *A policy* $\sigma$ *is* winning *for* $\varphi$ *from belief* $\mathfrak{b}$ *in (PO)MDP* $\mathcal{P}$ *iff* $\mathit{Pr}^{\sigma}_{\mathfrak{b}}(\mathit{AVOID}) = 0$ *and* $\mathit{Pr}^{\sigma}_{\mathfrak{b}}(\mathit{REACH}) = 1$*, i.e., if it reaches AVOID with probability zero and REACH with probability one (almost-surely) when* $\mathfrak{b}$ *is the initial state. Belief* $\mathfrak{b}$ *is* winning *for* $\varphi$ *in* $\mathcal{P}$ *if there exists a winning policy from* $\mathfrak{b}$*.*

We omit $\mathcal{P}$ and $\varphi$ whenever it is clear from the context and simply call $\mathfrak{b}$ winning.

**Problem 1:** Given a POMDP, a belief $\mathfrak{b}$, and a specification $\varphi$, decide whether $\mathfrak{b}$ is winning and find a policy $\sigma$ that is winning from $\mathfrak{b}$.

The problem is EXPTIME-complete [18]. Contrary to MDPs, it is not sufficient to consider memoryless policies.

Model checking queries for POMDPs often rely on the analysis of the *belief MDP*. Indeed, we may analyse this generally infinite model. Let us first recap a formal definition of the belief MDP, using the presentation from [11]. In the following, let $\mathbf{P}(s,\alpha,z) := \sum_{s' \in S} [\text{obs}(s')=z] \cdot \mathbf{P}(s,\alpha,s')$ denote the probability[1] to move to (a state with) observation $z$ from state $s$ using action $\alpha$. Then, $\mathbf{P}(\mathfrak{b},\alpha,z) := \sum_{s \in S} \mathfrak{b}(s) \cdot \mathbf{P}(s,\alpha,z)$ is the probability to observe $z$ after taking $\alpha$ in $\mathfrak{b}$. We define the *belief obtained by taking $\alpha$ from $\mathfrak{b}$, conditioned on observing $z$*:

$$\mathit{update}(\mathfrak{b}|\alpha,z)(s') := \frac{[\text{obs}(s')=z] \cdot \sum_{s \in S} \mathfrak{b}(s) \cdot \mathbf{P}(s,\alpha,s')}{\mathbf{P}(\mathfrak{b},\alpha,z)}. \tag{1}$$

**Definition 5 (Belief MDP).** *The* belief MDP *of POMDP $\mathcal{P} = \langle \mathcal{M}, \Omega, \text{obs} \rangle$ where $\mathcal{M} = \langle S, \text{Act}, \mu_{\text{init}}, \mathbf{P} \rangle$ is the MDP $\mathit{BelMDP}(\mathcal{P}) := \langle \mathcal{B}, \text{Act}, \mathbf{P}_{\mathcal{B}}, \mu_{\text{init}} \rangle$ with $\mathcal{B} = \mathit{Distr}(S)$, and transition function $\mathbf{P}_{\mathcal{B}}$ given by*

$$\mathbf{P}_{\mathcal{B}}(\mathfrak{b},\alpha,\mathfrak{b}') := \begin{cases} \mathbf{P}(\mathfrak{b},\alpha,\text{obs}(\mathfrak{b}')) & \text{if } \mathfrak{b}' = \mathit{update}(\mathfrak{b}|\alpha,\text{obs}(\mathfrak{b}')), \\ 0 & \text{otherwise.} \end{cases}$$

Due to (1) and the unique initial observation, we may restrict the beliefs to $\mathcal{B} = \bigcup_{z \in \Omega} \mathit{Distr}(\{s \mid \text{obs}(s) = z\})$, that is, each belief state has a unique associated observation. We can lift specifications to belief MDPs: *Avoid-beliefs* are the set of beliefs $\mathfrak{b}$ such that $\mathit{supp}(\mathfrak{b}) \cap \mathit{AVOID} \neq \emptyset$, and *reach-beliefs* are the set of beliefs $\mathfrak{b}$ such that $\mathit{supp}(\mathfrak{b}) \subseteq \mathit{REACH}$.

Towards obtaining a finite abstraction, the main algorithmic idea is the following. For the qualitative reach-avoid specifications we consider, the belief probabilities are irrelevant—*only the belief support is important* [47].

**Lemma 1.** *For winning belief $\mathfrak{b}$, belief $\mathfrak{b}'$ with $\mathit{supp}(\mathfrak{b}) = \mathit{supp}(\mathfrak{b}')$ is winning.*

Consequently, we can abstract the belief MDP into a finite belief support MDP.

**Definition 6 (Belief-Support MDP).** *For a POMDP $\mathcal{P} = \langle \mathcal{M}, \Omega, \text{obs} \rangle$ with $\mathcal{M} = \langle S, \text{Act}, \mu_{\text{init}}, \mathbf{P} \rangle$, the finite state space of a* belief-support MDP *$\mathcal{P}_B$ is $B = \big\{ b \subseteq S \mid \forall s, s' \in b \colon \text{obs}(s) = \text{obs}(s') \big\}$ where each state is the support of a belief state. Action $\alpha$ in state $b$ leads (with an irrelevant positive probability $p > 0$) to a state $b'$, if*

$$b' \in \Big\{ \bigcup_{s \in b} \text{post}_s(\alpha) \cap \{s \mid \text{obs}(s) = z\} \mid z \in \Omega \Big\}.$$

[1] We use Iverson brackets: $[x] = 1$ if $x$ holds and 0 otherwise.

Thus, transitions between states within $b$ and $b'$ are mimicked in the POMDP. Equivalently, the following clarifies the belief-support MDP as an abstraction of the belief MDP: there are transitions with action $\alpha$ between $b$ and $b'$, if there exists beliefs $\mathfrak{b}, \mathfrak{b}'$ with $\mathit{supp}(\mathfrak{b}) = b$ and $\mathit{supp}(\mathfrak{b}') = b'$, such that $\mathfrak{b}' \in \text{post}_{\mathfrak{b}}(\alpha)$. We lift the specification as before:

**Definition 7 (Lifted specification).** *For $\varphi = \langle \mathit{AVOID}, \mathit{REACH} \rangle$, we define $\varphi_B = \langle \mathit{AVOID}_B, \mathit{REACH}_B \rangle$ with $\mathit{AVOID}_B = \{b \mid b \cap \mathit{AVOID} \neq \emptyset\}$, and $\mathit{REACH}_B = \{b \mid b \subseteq \mathit{REACH}\}$.*

We obtain the following lemma, which follows from the fact that almost-sure reachability is a graph property[2].

**Lemma 2.** *If belief $\mathfrak{b}$ is winning in the POMDP $\mathcal{P}$ for $\varphi$, then the support $\mathit{supp}(\mathfrak{b})$ is winning in the belief-support MDP $\mathcal{P}_B$ for $\varphi_B$.*

Lemma 2 yields an equivalent reformulation of Problem 1 for belief supports:

**Problem 1 (equivalent):** Given a POMDP $\mathcal{P}$, belief $\mathfrak{b}$, and specification $\varphi$, decide whether $\mathit{supp}(\mathfrak{b})$ is winning for $\varphi_B$ in the belief-support MDP $\mathcal{P}_B$.

## 3 Winning Regions

This section provides the observations on winning regions, a key concept for this paper. An important consequence of Lemma 2 and the reformulation of Problem 1 to the belief-support MDP is that the initial distribution of the POMDP is no longer relevant. Winning policies for individual beliefs may be composed to a policy that is winning for all of these beliefs, using the individual action choices.

**Lemma 3.** *If the policies $\sigma$ and $\sigma'$ are winning for the belief supports $b$ and $b'$, respectively, then there exists a policy $\sigma''$ that is winning for both $b$ and $b'$.*

While this statement may seem trivial on the MDP (or equivalently on beliefs), we notice that it does not hold for POMDP states. As a natural consequence, we are able to consider winning beliefs without referring to a specific policy.

**Definition 8 (Winning region).** *Let $\sigma$ be a policy. A set $W_\varphi^\sigma \subseteq B$ of belief supports is a* winning region for $\varphi$ and $\sigma$, *if $\sigma$ is winning from each $b \in W_\varphi^\sigma$. A set $W_\varphi \subseteq B$ is a winning region for $\varphi$, if every $b \in W_\varphi$ is winning. The region containing all winning beliefs is the* maximal winning region[3].

[2] Although the probabilities are not relevant to compute almost-sure reachability, it is important to notice that almost-sure reachability is different from sure-reachability [5]: For almost-sure reachability, there can be an infinite path that never reaches the target, as long as the probability mass over all those paths is 0. Almost-sure reachability can, however, be expressed as sure-reachability in a particular game-setting [47].

[3] In some literature, *winning region* always refers to a *maximal* winning region.

Observe that the maximal winning region in MDPs exists for qualitative reachability, but not for quantitative reachability, which we do not consider here.

**Problem 2:** Given a POMDP $\mathcal{P}$ and a specification $\varphi$, find the maximal winning region $W_\varphi$.

Using this definition of winning regions, we are able to reformulate **Problem 1** by asking whether the support of some belief $\mathfrak{b}$ is in the winning region.

Part of **Problem 1** was to compute a winning policy. Below, we study the connection between the winning region and winning policies. We are interested in subsets of the maximal winning region that exhibit two properties:

**Definition 9 (Deadlock-free).** *A set $W$ of belief-supports $W \subseteq B$ is* deadlock-free*, if for every $b \in W$, an action $\alpha \in \mathrm{EnAct}(b)$ exists such that $\mathrm{post}_b(\alpha) \subseteq W$.*

**Definition 10 (Productive).** *A set of belief supports $W \subseteq B$ is* productive *(towards a set $REACH_B$), if from every $b \in W$, there exists a (finite) path $\pi = b_0\alpha_1 b_1 \dots b_n$ from $b_0$ to $b_n \in REACH_B$ with $b_i \in W$ and $\mathrm{post}_{b_i}(\alpha) \subseteq W$ for all $1 \leq i \leq n$.*

Every productive region is deadlock-free, as *REACH*-states are absorbing. The maximal winning region is productive towards $REACH_B$ (and thus deadlock-free) by definition. Intuitively, while a deadlock-free region ensures that one never has to leave the region, any productive winning region ensures that from every belief support within this region there is a policy to stay in the winning region and that can almost-surely reach a *REACH*-state. In particular, to find a winning policy (Challenge 1) or for the purpose of safe exploration (Challenge 2), it is sufficient to find a productive subset of the maximal winning region. We detail on this insight in Sect. 6.

**Problem 3:** Given a POMDP $\mathcal{P}$ and a specification $\varphi$, find a (large) productive winning region $W_\varphi$.

To allow a compact representation of winning regions, we exploit that for any belief support $b' \subseteq b$ it holds that $\mathrm{post}_{b'}(\alpha) \subseteq \mathrm{post}_b(\alpha)$ for all actions $\alpha \in \mathrm{Act}$, that is, the successors of $b'$ are contained in the successors of $b$.

**Lemma 4.** *For winning belief support $b$, $b' \subseteq b$ is winning.*

## 4 Iterative SAT-Based Computation of Winning Regions

We devise an approach for iteratively computing an increasing sequence of productive winning regions. The approach delivers a compact symbolic encoding of winning regions: For a belief (or belief-support) state from a given winning region, we can efficiently decide whether the outcome of an action emanating from the state stays within the winning region.

Key ingredient is the computation of so-called memoryless winning policies. We start this section by briefly recapping how to compute such policies directly

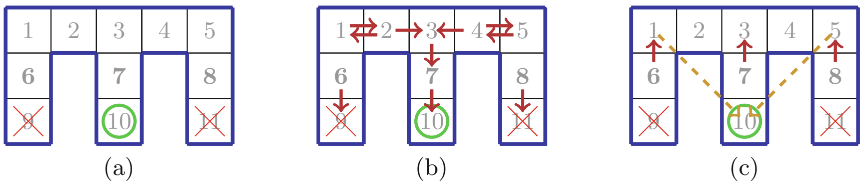


**Fig. 1.** Cheese-Maze example to explain memoryless policies and shortcuts

on the POMDP, before we build an efficient incremental approach on top of this base method. In particular, we first present a naive iterative algorithm based on the notion of *shortcuts*, then describe how to implicitly add shortcuts within the encoding, and then finally combine the ideas to an efficient algorithm.

## 4.1 One-Shot Approach to Find Small Policies from a Single Belief

We aim to solve **Problem 1** and determine a winning policy. The number of policies is exponential in the actions and the (exponentially many) belief support states. Searching among doubly exponentially many possibilities is intractable in general. However, Chatterjee et al. [15] observe that often much simpler winning policies exist and provides a *one-shot approach* to find them. The essential idea is to search only for memoryless observation-based policies $\sigma\colon \Omega \to \mathit{Distr}(\mathrm{Act})$ that are winning for the (initial) belief support $b$.

*Example 1.* Consider the small Cheese-POMDP [35] in Fig. 1(a). States are cells, actions are moving in the cardinal directions (if possible), and observations are the directions with adjacent cells, e.g., the boldface states 6, 7, 8 share an observation. We set $\mathit{REACH} = \{10\}$ and $\mathit{AVOID} = \{9, 11\}$. From belief support $b = \{6, 8\}$ there is no memoryless winning policy—In states $\{6, 8\}$ we have to go north, which prevents us from going south in state 7. However, we can find a memoryless winning policy for $\{1, 5\}$, see Fig. 1(b).

This problem is NP-complete, and it is thus natural to encode the problem as a satisfiability query in propositional logic. We mildly adapt the original encoding of winning policies [15]. We introduce three sets of Boolean variables: $A_{z,\alpha}$, $C_s$ and $P_{s,j}$. If a policy takes action $\alpha \in \mathrm{Act}$ with positive probability upon observation $z \in \Omega$, then and only then, $A_{z,\alpha}$ is `true`. If under this policy a state $s \in S$ is reached from some initial belief support $b_\iota$ with positive probability, then and only then, $C_s$ is `true`. We define a maximal rank $k$ to ensure the productivity. For each state $s$ and rank $0 \leq j \leq k$, variable $P_{s,j}$ indicates rank $j$ for $s$, that is, a path from $s$ leads to $s' \in \mathit{REACH}$ within $j$ steps.[4] A winning policy is then obtained by finding a satisfiable solution (via a SAT solver) to the conjunction $\Psi^{\varphi}_{\mathcal{P}}(b_\iota, k)$ of the constraints (2a)–(5), where $S_? = S \setminus \big(\mathit{AVOID} \cup \mathit{REACH}\big)$.

[4] Notice that a state $s$ can have multiple 'ranks' in this encoding. Its rank is the smallest $j$ such that $P_{s,j}$ is `true`.

$$\bigwedge_{s\in b_\iota} C_s \qquad (2a) \qquad\qquad \bigwedge_{z\in\Omega} \Big( \bigvee_{\alpha\in\mathrm{EnAct}(z)} A_{z,\alpha}\Big) \qquad (2b)$$

The initial belief support is clearly reachable (2a). The conjunction in (2b) ensures that in every observation, at least one action is taken.

$$\bigwedge_{s\in AVOID} \neg C_s \quad\wedge \bigwedge_{\substack{s\in S\\ \alpha\in\mathrm{EnAct}(s)}} \Big(C_s \wedge A_{\mathrm{obs}(s),\alpha} \rightarrow \bigwedge_{s'\in\mathrm{post}_s(\alpha)} C_{s'}\Big) \tag{3}$$

The conjunction (3) ensures that for any model for these formulas, the set of states $\{s \in S \mid C_s = \mathtt{true}\}$ is reachable, does not overlap with *AVOID*, and is transitively closed under reachability (for the policy described by $A_{z,\alpha}$).

$$\bigwedge_{s\in S_?} C_s \rightarrow P_{s,k} \tag{4}$$

$$\bigwedge_{s\notin REACH} \neg P_{s,0} \quad\wedge \bigwedge_{\substack{s\in S_?\\ 1\leq j\leq k}} P_{s,j} \leftrightarrow \Big( \bigvee_{\alpha\in\mathrm{EnAct}(s)} \big(A_{\mathrm{obs}(s),\alpha} \wedge \big( \bigvee_{s'\in\mathrm{post}_s(\alpha)} P_{s',j-1}\big)\big)\Big) \tag{5}$$

Conjunction (4) states that any state that is reached almost-surely reaches a state in *REACH*, i.e., that there is a path (of length at most) $k$ to the target. Conjunctions (5) describe a ranking function that ensures the existence of this path. Only states in *REACH* have rank zero, and a state with positive probability to reach a state with rank $j-1$ within a step has rank at most $j$.

By [15, Thm. 2], it holds that the conjunction $\Psi^{\varphi}_{\mathcal{P}}(b_\iota, k)$ of the constraints (2a)–(5) is satisfiable, if there is a memoryless observation-based policy such that $\varphi$ is satisfied. If $k = |S|$, then the reverse direction also holds. If $k < |S|$, we may miss states with a higher rank. Large values for $k$ are practically intractable [15], as the encoding grows significantly with $k$. Pandey and Rintanen [41] propose extending SAT-solvers with a dedicated handling of ranking constraints.

In order to apply this to small-memory policies, one can unfold $\log(m)$ bits of memory of such a policy into an $m$ times larger POMDP [15,33], and then search for a memoryless policy in this larger POMDP. Chatterjee et al. [15] include a slight variation to this unfolding, allowing smaller-than-memoryless policies by enforcing the same action over various observations.

## 4.2 Iterative Shortcuts

We exploit the one-shot approach to create a naive iterative algorithm that constructs a productive winning region. The iterative algorithm avoids the following restrictions of the one-shot approach. (1) In order to increase the likelihood of finding winning policies, we do not restrict ourselves to small-memory policies, and (2) we do not have to fix a maximal rank $k$. These modifications allow us to find more winning policies, without guessing hyper-parameters. As we do not need to fix the belief-state, those parts of the winning region that are easy to find for the solver are encountered first.

*The One-Shot Approach on Winning Regions.* To understand the naive iterative algorithm, it is helpful to consider the previous encoding in the light of **Problem 3**, i.e., finding productive winning regions. Consider first the interpretation of the variables. Indeed, observe that we have found *the same* winning policy for all states $s$ where $C_s$ is `true`. Consequentially, any belief support $b_z = \{s \mid C_s\ \texttt{true} \wedge \mathrm{obs}(s) = z\}$ is winning.

**Lemma 5.** *If $\sigma$ is winning for $b$ and $b'$, then $\sigma$ is also winning for $b \cup b'$.*

This lemma is somewhat dual to Lemma 4, but requires a fixed policy. The constraints (3) and ensure that a winning-region is deadlock-free. The constraints (4) and (5) ensure productivity of the winning region.

*Adding Shortcuts Explicitly.* The key idea is that we iteratively add *short-cuts* in the POMDP that represent known winning policies. We find a winning policy $\sigma$ for some belief states in the first iteration, and then add a fresh action $\alpha_\sigma$ to all (original) POMDP states: This action leads – with probability one – to a *REACH* state, if the state is in the wining belief-support under policy $\sigma$. Otherwise, the action leads to an *AVOID* state.

**Definition 11.** *For POMDP $\mathcal{P} = \langle \mathcal{M}, \Omega, \mathrm{obs}\rangle$ where $\mathcal{M} = \langle S, \mathrm{Act}, \mu_{\mathrm{init}}, \mathbf{P}\rangle$ and a policy $\sigma$ with associated winning region $W_\varphi^\sigma$, and assuming w.l.o.g., $\top \in$ REACH and $\bot \in$ AVOID, we define the* shortcut POMDP $\mathcal{P}\{\sigma\} = \langle \mathcal{M}', \Omega, \mathrm{obs}\rangle$ *with $\mathcal{M}' = \langle S, \mathrm{Act}', \mu_{\mathrm{init}}, \mathbf{P}'\rangle$, $\mathrm{Act}' = \mathrm{Act} \cup \{\alpha_\sigma\}$, $\mathbf{P}'(s, \alpha) = \mathbf{P}(s, \alpha)$ for all $s \in S$ and $\alpha \in \mathrm{Act}$, and $\mathbf{P}'(s, \alpha_\sigma) = \{\top \mapsto [\{s\} \in W_\varphi^\sigma], \bot \mapsto [\{s\} \notin W_\varphi^\sigma]\}$.*

**Lemma 6.** *For a POMDP $\mathcal{P}$ and policy $\sigma$, the (maximal) winning regions for $\mathcal{P}\{\sigma\}$ and $\mathcal{P}$ coincide.*

First, adding more actions will not change a winning belief-support to be not winning. Furthermore, by construction, taking the novel action will only lead to a winning belief-support whenever following $\sigma$ from that point onwards would be a winning policy. The *key* benefit is that adding shortcuts may extend the set of belief-support states that win via a memoryless policy. This observation also gives rise to the following extension to the one-shot approach.

*Example 2.* We continue with Example 1. If we add shortcuts, we can now find a memoryless winning policy for $b = \{6, 8\}$, depicted in Fig. 1(c).

*Iterative Shortcuts to Extend a Winning Region.* The idea is now to run the one-shot approach, extract the winning region, add the shortcuts to the POMDP, and rerun the one-shot approach. To make the one-shot approach applicable in this setting, it only needs one change: Rather than fixing an initial belief-support, we ask for an arbitrary new belief-support to be added to the states that we have previously covered. We use a data structure $\mathsf{Win}$ such that $\mathsf{Win}(z)$ encodes all winning belief supports with observation $z$. Internally, the data structure stores maximal winning belief supports (w.r.t. set inclusion, see also Lemma 4) as bit-vectors. By construction, for every $b \in \mathsf{Win}(z)$, a winning region exists, i.e., conceptually, there is a shortcut-action leading to *REACH*.

**Algorithm 1** Naive construction of winning regions

```
Input: POMDP $\mathcal{P}$, reach-avoid specification $\varphi$
Output: Winning region encoded in $\mathsf{Win}$
$\mathsf{Win}(z) \leftarrow \{s \in \mathit{REACH} \mid \text{obs}(s) = z\}$ for all $z \in \Omega$
$\Phi \leftarrow \mathit{Encode}(\mathcal{P}, \varphi, \mathsf{Win})$                ▷ Create encoding (2b),(3),(6),(7).
while $\exists \eta$ s.t. $\eta \models \Phi$ do                                  ▷ Call an SMT solver
    $\mathsf{Win}(z) \leftarrow \mathsf{Win}(z) \cup \{b \mid s \in b \text{ iff } \eta(C_s)\}$ for all $z \in \Omega$
    $\mathcal{P} \leftarrow \mathcal{P}\{\sigma_\eta\}$                               ▷ Extend POMDP with Def. 11
                                                                 ▷ with $\sigma_\eta$ policy encoded by $\eta$.
    $\Phi \leftarrow \mathit{Encode}(\mathcal{P}, \varphi, \mathsf{Win})$
```

We extend the encoding (in partial preparation of the next subsection) and add a variable $U_z \in b$ that is `true` if the policy is winning in a belief support that is not yet in $\mathsf{Win}(z)$. We replace (2a) with:

$$\bigvee_{z \in \Omega} U_z \quad \wedge \bigwedge_{\substack{z \in \Omega \\ \mathsf{Win}(z)=\emptyset}} \Big( U_z \leftrightarrow \bigvee_{\substack{s \in S \\ \text{obs}(s)=z}} C_s \Big) \quad \wedge \bigwedge_{\substack{z \in \Omega \\ \mathsf{Win}(z)\neq\emptyset}} \Big( U_z \leftrightarrow \bigwedge_{X \in \mathsf{Win}(z)} \bigvee_{\substack{s \in S \setminus X \\ \text{obs}(s)=z}} C_s \Big) \tag{6}$$

For an observation $z$ for which we have not found a winning belief support yet, finding a policy from any state $s$ with obs($s$) updates the winning region. Otherwise, it means finding a winning policy for a belief support that is not subsumed by a previous one (6).

*Real-Valued Ranking.* To avoid setting a maximal path length, we use unbounded (real) variables $R_s$ rather than Boolean variables for the ranking [57]. This relaxation avoids the growth of the encoding and admits arbitrarily large ranks with a fixed-size encoding into difference logic. This logic is an extension to propositional logic that can be checked using an SMT solver [6].

$$\bigwedge_{s \in S_?} C_s \rightarrow \Big( \bigvee_{\alpha \in \text{EnAct}(s)} \big( A_{\text{obs}(s),\alpha} \wedge \big( \bigvee_{s' \in \text{post}_s(\alpha)} R_s > R_{s'} \big)\big)\Big) \tag{7}$$

We replace (4) and (5): A state must have a successor state with a lower rank – as before, but with real-valued ranks (7).

*Algorithm.* Together, the algorithm is given in Algorithm 1. We initialize the winning region based on the specification, then encode the POMDP using the (modified) one-shot encoding. As long as the SMT solver finds policies that are winning for a new belief-support, we add those belief supports to the winning region. In each iteration, Win contains a winning region. Once we find no more policies that extend the winning region on the extended POMDP, we terminate.

The algorithm always terminates because the set of winning regions is finite, but in general does not solve **Problem 2**. Formally, the maximal winning region is a greatest fixpoint [5] and we iterate from below, i.e., the fixpoint that we find will be the smallest fixpoint (of the operation that we implement). However, iterating from above requires to reason that none of the doubly-exponentially many policies is winning for a particular belief support state; whereas our approach profits from finding simple strategies early on. Unfolding of memory as discussed earlier also makes this algorithm complete, yet, suffers from the same blow-up. A main advantage is that the algorithm often avoids the need for unfolding when searching for a winning policy or large winning regions.

Next, we address two weaknesses: First, the algorithm currently creates a new encoding in every iteration, yielding significant overhead. Second, the algorithm in many settings requires adding a bit of memory to realize behavior where in a particular observation, we *first* want to execute an action $\alpha$ and *then* follow a shortcut from the state (with the same observation) reached from there. We adapt the encoding to explicitly allow for these (non-memoryless) policies.

## 4.3 Incremental Encoding of Winning Regions

In this section, instead of naively adjusting the POMDP, we realize the idea of adding shortcuts directly on the encoding. This encoding is the essential step towards an efficacious approach for solving **Problem 3**. We find winning states based on a previous solution, and instead of adding actions, we allow the solver to decide following individual policies from each observation. In Sect. 4.4, we embed this encoding into an improved algorithm.

Our encoding represents an observation-based policy that can decide to take a shortcut, which means that it follows a previously computed winning policy from there (implicitly using Lemma 3). In addition to $A_{z,\alpha}$, $C_s$ and $R_s$ from the previous encoding, we use the following variables: The policy takes shortcuts in states $s$ where $D_s$ is `true`. For each observation, we must take the same shortcut, referred to by a positive integer-valued index $I_z$. More precisely, $I_z$ refers to a shortcut from a previously computed (fragment of a) winning region stored in $\mathsf{Win}(z)_{I_z}$. The policy may decide to *switch*, that is, to follow a shortcut *after* taking an action starting in a state with observation $z$. If $F_z$ is `true`, the policy takes some action from $z$-states and from the next state, we take a shortcut. The encoding thus implicitly represents policies that are not memoryless but rather allow for a particular type of memory.
The conjunction of (6) and (8)–(13) yields the encoding $\Phi^{\varphi}_{\mathcal{P}}(\mathsf{Win})$:

$$\bigwedge_{z\in\Omega}\Big(\bigvee_{\alpha\in\mathrm{EnAct}(z)} A_{z,\alpha}\Big) \quad\wedge \bigwedge_{s\in AVOID} \neg C_s \wedge \neg D_s \tag{8}$$

$$\bigwedge_{\substack{s\in S\\ \alpha\in\mathrm{EnAct}(s)}} \Big(C_s \wedge A_{\mathrm{obs}(s),\alpha} \wedge \neg F_{\mathrm{obs}(s)} \quad\rightarrow \bigwedge_{s'\in\mathrm{post}_s(\alpha)} C_{s'}\Big) \tag{9}$$

$$\bigwedge_{\substack{s\in S\\ \alpha\in\mathrm{EnAct}(s)}} \Big(C_s \wedge A_{\mathrm{obs}(s),\alpha} \wedge F_{\mathrm{obs}(s)} \quad\rightarrow \bigwedge_{s'\in\mathrm{post}_s(\alpha)} D_{s'}\Big) \tag{10}$$

Similar to (2b), (3), we select at least one action and *AVOID*-states should not be reached (8). States reached are closed under the transitive closure, however,

**Algorithm 2** Naive construction of winning regions with incremental encoding

```
**Input**: POMDP $\mathcal{P}$, reach-avoid specification $\varphi$
**Output**: Winning region encoded in $\mathsf{Win}$
$\mathsf{Win}(z) \leftarrow \{s \in \mathit{REACH} \mid \mathrm{obs}(s) = z\}$ for all $z \in \Omega$
$\Phi \leftarrow \mathit{Encode}(\mathcal{P}, \varphi, \mathsf{Win})$        ▷ Create encoding (6),(8)–(13).
**while** $\exists \eta$ s.t. $\eta \models \Phi$ **do**        ▷ Call an SMT solver
    $\mathsf{Win}(z) \leftarrow \mathsf{Win}(z) \cup \{b \mid s \in b \text{ iff } \eta(C_s)\}$ for all $z \in \Omega$
    $\Phi \leftarrow \mathit{Encode}(\mathcal{P}, \varphi, \mathsf{Win})$
```

only if we do not switch to taking a shortcut (9). Furthermore, we mark the states reached after switching (10) and need to select a shortcut for these states.

$$\bigwedge_{s \in S} \big(D_s \;\rightarrow\; I_{\mathrm{obs}(s)} > 0\big) \quad\wedge\quad \bigwedge_{z \in \Omega} I_z \leq |\mathsf{Win}(z)| \tag{11}$$

$$\bigwedge_{\substack{z \in \Omega \\ 0 < i \leq |\mathsf{Win}(z)|}} \quad \bigwedge_{\substack{s \in S \setminus \mathsf{Win}(z)_i \\ \mathrm{obs}(s) = z}} D_s \;\rightarrow\; I_z \neq i \tag{12}$$

If we reach a state $s$ after switching, then we must pick a shortcut. We can only pick an index that reflects a found winning region (11). If we pick this shortcut reflecting a winning region (fragment) for observation $z$, then we are winning from the states in $\mathsf{Win}(z)_i$, but not from any other state $s$ with that observation. Thus, for $s \notin \mathsf{Win}(z)_i$, if we are going to follow any shortcut (that is, $D_s$ holds), we should not pick this particular shortcut encoded by $I_z$ (because it will lead to an *AVOID*-state). In terms of the policy: Taking this previously computed policy from state $s$ is not (known to) lead us to a *REACH*-state (12). Finally, we update the ranking to account for shortcuts.

$$\bigwedge_{s \in S_?} C_s \rightarrow \Big( \bigvee_{\alpha \in \mathrm{EnAct}(s)} \big(A_{\mathrm{obs}(s),\alpha} \wedge \big( \bigvee_{s' \in \mathrm{post}_s(\alpha)} R_s > R_{s'}\big)\big) \vee F_{\mathrm{obs}(s)} \Big) \tag{13}$$

We make a slight adaption to (7): Either we have a successor state with a lower rank (as before) or we follow a shortcut—which either leads to the target or to violating the specification (13). We formalize the correctness of the encoding:

**Lemma 7.** *If $\eta \models \Phi^{\varphi}_{\mathcal{P}}(\mathsf{Win})$, then for every observation $z$, the belief support $b_z = \{s \mid \eta(C_s) = \mathtt{true}, \mathrm{obs}(s) = z\}$ is winning.*

Algorithm 2 is a straightforward adaption of Algorithm 1 that avoids adding shortcuts explicitly (and uses the updated encoding). As before, the algorithm terminates and solves **Problem 3**. We conclude:

**Theorem 1.** *In any iteration, Algorithm 2 computes a productive winning region.*

### 4.4 An Incremental Algorithm

We adapt the algorithm sketched above to exploit the incrementality of modern SMT solvers. Furthermore, we aim to reduce the invocations of the solver by finding some extensions to the winning region via a graph-based algorithm.

**Algorithm 3** Incremental construction of winning regions

```
Input: POMDP $\mathcal{P}$, reach-avoid specification $\varphi$
Output: Winning region encoded in $\mathsf{Win}$
$\mathsf{Win}(z) \leftarrow \{s \in \mathit{REACH} \mid \mathrm{obs}(s) = z\}$ for all $z \in \Omega$
$\mathsf{Win} \leftarrow \mathit{GraphPreprocessing}(\mathsf{Win})$
$\Phi_{\mathrm{fix}} \leftarrow \mathit{Encode}_{\mathrm{fix}}(\mathcal{P}, \varphi, \mathsf{Win})$          ▷ Create encoding (8)–(13)
$\Phi_{\mathrm{inc}} \leftarrow \mathit{Encode}_{\mathrm{inc}}(\mathcal{P}, \varphi, \mathsf{Win})$          ▷ Encode (6)
while $\exists \eta$ s.t. $\eta \models \Phi_{\mathrm{fix}} \wedge \Phi_{\mathrm{inc}}$ do          ▷ Call an SMT solver, fix $\eta$
    do          ▷ Extend policy
        $\Phi_\eta \leftarrow \bigwedge\{A_{z,\alpha} \mid \eta(U_z) \wedge \eta(A_{z,\alpha})\}$          ▷ Part. fix policy
    while $\exists \eta$ s.t. $\eta \models \Phi_{\mathrm{fix}} \wedge \Phi_{\mathrm{var}} \wedge \Phi_\eta$          ▷ Call SMT, fix $\eta$
    $\mathsf{Win}(z) \leftarrow \mathsf{Win}(z) \cup \{B \mid s \in B \text{ iff } \eta(C_s)\}$ for all $z \in \Omega$
    $\mathsf{Win} \leftarrow \mathit{GraphPreprocessing}(\mathsf{Win})$
    $\Phi_{\mathrm{fix}} \leftarrow \Phi_{\mathrm{fix}} \wedge \mathit{Encode}_{(11)(12)}(\mathcal{P}, \varphi, \mathsf{Win})$          ▷ Update: (11),(12)
    $\Phi_{\mathrm{inc}} \leftarrow \mathit{Encode}_{\mathrm{inc}}(\mathcal{P}, \varphi, \mathsf{Win})$          ▷ Encode (6)
```

*Graph-Based Preprocessing.* To reduce the number of SMT invocations, we employ polynomial-time graph-based heuristics. The first step is to use (fully observable) MDP model checking on the POMDP as follows: find all states that under each (not necessarily observation-based) policy reach an *AVOID*-state with positive probability, and make them absorbing. Then, we find all states that under *each* policy reach a *REACH*-state almost-surely. Then, we iteratively search for *winning observations* and use them to extend the *REACH*-states. An observation $z$ is winning, if the belief-support $\{s \mid \mathrm{obs}(s) = z\}$ is winning. We start with a previously determined winning region $W$. We iteratively update $W$ by adding states $b_z = \{s \mid \mathrm{obs}(s) = z\}$ for some observation $z$, if there is an action $\alpha$ such that from every $s \in b_z$, it holds $\mathrm{post}_s(\alpha) \subseteq W$. The iterative updates are interleaved with MDP model checking on the POMDP as described above until we find a fixpoint.

*Optimized Algorithm.* We improve Algorithm 2 along four dimensions to obtain Algorithm 3. First, we employ fewer updates of the winning region: We aim to extend the policy as much as possible, i.e., we want the SMT-solver to find more states with the same observation that are winning under the same policy. Therefore, we fix the variables for action choices that yield a new winning policy, and let the SMT solver search whether we can extend the corresponding winning region by finding more states and actions that are compatible with the partial policy. Second, we observe that between (outer) iterations, large parts of the encoding stay intact, and use an incremental approach in which we first push all the constraints from the POMDP onto the stack, then all the constraints from the winning region, and finally a constraint that asks for progress. After we found a new policy, we pop the last constraint from the stack, add new constraints regarding the winning region (notice that the old constraints remain intact), and push new constraints that ask for extending the winning region to the stack. We refresh the encoding periodically to avoid unnecessary cluttering. Third, further constraints (1) make the usage of shortcuts more flexible—we allow taking shortcuts either immediately or after the next action, and (2) enable an even more incremental encoding with some minor technical reformulations. Fourth, we add the graph-preprocessing discussed above during the outer iteration.

## 5 Symbolic Model Checking for the Belief-Support MDP

In this section, we briefly describe how we encode a given POMDP into a belief-support MDP to employ symbolic, off-the-shelf probabilistic model checking. In particular, we employ symbolic (decision-diagram, DD) representations of the belief-support MDP as we expect this MDP to be huge. Constructing that DD representation effectively is not entirely trivial. Instead, we advocate constructing a (modular) symbolic description of the belief support MDP. Concretely, we automatically generate a model description in the MDP modeling language JANI [13],[5] and then apply off-the-shelf model checking on the JANI description.

Conceptually, we create a belief-support MDP with auxiliary states to allow for a concise encoding.[6] We use this auxiliary state $\hat{b}$ to describe for any transition the conditioning on the observation. Concretely, a single transition $\mathbf{P}(b, \alpha, b')$ in the belief-support MDP is reflected by two transitions $\mathbf{P}(b, \alpha, \hat{b})$ and $\mathbf{P}(\hat{b}, \alpha_\bot, b')$ in our encoding, where $\alpha_\bot$ is a unique dummy action. We encode states using triples $\langle \texttt{belsup}, \texttt{newobs}, \texttt{lact} \rangle$. `belsup` is a bit vector with entries for every state $s$ that we use to encode the belief support. Variables `newobs` and `lact` store an observation and an action and are relevant only for the auxiliary states. Technically, we now encode the first transition from $b$ with the nondeterministic action $\alpha$ to $\hat{b}$. $\mathbf{P}(b, \alpha)$ then yields (with arbitrary positive) probability a new observation that will reflect the observation $\text{obs}(b')$. We store $\alpha$ and $\text{obs}(b')$ in `lact` and `newobs`, respectively. The second step is a single deterministic (dummy) action updating `belsup` while taking into account `newobs`. The step also resets `lact` and `newobs`.

The encoding of the transitions as follows: For the first step, we create nondeterministic choices for each action $\alpha$ and observation $z$. We guard these choices with $z$ meaning that the edge is only applicable to states having observation $z$, i.e., the guard is $\bigvee_{s \in S, \text{obs}(s)=z} \texttt{belsup}(s)$. With these guarded edges, we define the destinations: With an arbitrary[7] probability $p$, we go to an observation $z_1$ *if* there is at least one state in $s \in \texttt{belsup}$ which has a successor state $s' \in \text{post}_s(\alpha)$ with $\text{obs}(s') = z_1$.

[5] The description here works on a network of synchronized state machines as is also common in the PRISM language.

[6] The usage of message passing or *indexed assignments* in JANI would circumvent the need for intermediate states, but is to the best of our knowledge not supported by decision-diagram based model checkers.

[7] We leave this a parametric probability in model building to reduce the number of different probabilities, as this is beneficial for the size of the decision diagram that Storm constructs – it will only have leafs 0, $p$, 1. Technically, such MDPs are not necessarily well-defined but we can employ model checking on the graph structure.

The following pseudocode reflects the first step in the transition encoding. The syntax is as follows: **take** an action **if** a Boolean guard is satisfied, then updates are executed with probability **prob**. An example for a guard is an observation $z$.

$$\textbf{take}\,\alpha\,\textbf{if}\,z\,\textbf{then}\begin{cases}\textbf{prob}\ \Big(\bigvee_{\substack{s\in S\\ \mathbf{P}(s,\alpha,z_1)>0}} \texttt{belsup}(s)\ ?\ p : 0\Big): & \begin{array}{c}\texttt{newobs} \leftarrow z_1\\ \texttt{lact} \leftarrow \alpha\end{array}\\ \dots & \dots \\ \textbf{prob}\ \Big(\bigvee_{\substack{s\in S\\ \mathbf{P}(s,\alpha,z_n)>0}} \texttt{belsup}(s)\ ?\ p : 0\Big): & \begin{array}{c}\texttt{newobs} \leftarrow z_n\\ \texttt{lact} \leftarrow \alpha\end{array}\end{cases}$$

The second step synchronously updates each state $s'$ in the POMDP independently: The entry $\texttt{belsup}(s')$ is set to $\texttt{true}$ if $\text{obs}(s) = \texttt{newobs}$ and if there is a state $s$ currently $\texttt{true}$ in (the old) $\texttt{belsup}$ with $s' \in \text{post}_s(\texttt{lact})$. The step thus can be captured by the following pseudocode for each $s'$:

$$\textbf{take}\,\alpha_\perp\,\textbf{if}\,\texttt{true}\,\textbf{then}\,\textbf{prob}\,1 : \texttt{belsup}(s') \leftarrow \Big(\bigvee_s \mathbf{P}(s,\texttt{lact},s') > 0\Big) \wedge \text{obs}(s')$$

Finally, whenever the dummy action $\alpha_\perp$ is executed, we also reset the variables $\texttt{newobs}$ and $\texttt{lact}$. The resulting encoding thus has transitions in the order of $|S| + |\Omega|^2 \cdot |\max_{z\in\Omega} \text{EnAct}(z)|$.

## 6 Almost-Sure Reachability Shields in POMDPs

In this section, we define a *shield* for POMDPs – towards the application of safe exploration (Challenge 2) – that blocks actions which would lead an agent out of a winning region. In particular, the shield imposes restrictions on policies to satisfy the reach-avoid specification. Technically, we adapt so-called *permissive* policies [21,31] for a belief-support MDP. To force an agent to stay within a productive winning region $W_\varphi$ for specification $\varphi$, we define a *$\varphi$-shield* $\nu\colon b \rightarrow 2^{\text{Act}}$ such that for any winning $b$ for $\varphi$ we have $\nu(b) \subseteq \{\alpha \in \text{Act} \mid \text{post}_b(\alpha) \subseteq W_\varphi\}$, i.e., an action is part of the shield $\nu(b)$ if it exclusively leads to belief support states within the winning region.

A shield $\nu$ restricts the set of actions an arbitrary policy may take[8]. We call such restricted policies *admissible*. Specifically, let $b_\tau$ be the belief support after observing an observation sequence $\tau$. Then policy $\sigma$ is $\nu$-admissible if $supp(\sigma(\tau)) \subseteq \nu(b_\tau)$ for every observation-sequence $\tau$. Consequently, a policy is *not* admissible if for some observation sequence $\tau$, the policy selects an action $\alpha \in \text{Act}$ which is not allowed by the shield.

Some admissible policies may choose to stay in the winning region without progressing towards the *REACH* states. Such a policy adheres to the avoid-part of the specification, but violates the reachability part. To enforce *progress*, we

[8] While memory policies based on the belief (support) are sufficient to ensure almost-sure reachability, the goal is to shield other policies that do not necessarily fall in this restricted class.

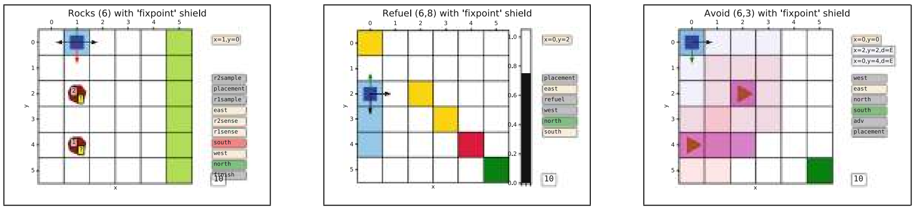


**Fig. 2.** Video stills from simulating a shielded agent on three different benchmarks.

adapt a notion of *fairness*. A policy is fair if it takes every action infinitely often at any belief support state that appears infinitely often along a trace [5]. For example, a policy that randomizes (arbitrarily) over all actions is fair–we notice that most reinforcement learning policies are therefore fair.

**Theorem 2.** *For a $\varphi$-shield $\nu$ and a winning belief support $b$, any* fair *$\nu$-admissible policy satisfies $\varphi$ from $b$.*

We give a proof (sketch) in [32, Appendix]. The main idea is to show that the induced Markov chain of any admissible policy has only bottom SCCs that contain *REACH*-states.

*Remark 1.* If $\varphi$ is a safety specification (where $Pr^{\sigma}_{\mathfrak{b}}(AVOID) = 0$ suffices), we can rely on deadlock-free winning regions rather than productive winning regions and drop the fairness assumption.

## 7 Empirical Evaluation

We investigate the applicability of our incremental approach (Algorithm 3) to **Challenge 1** and **Challenge 2**, and compare with our adaption and implementation of the one-shot approach [15], see Sect. 4.1. We also employ the MDP model-checking approach from Sect. 5. Experiments, videos, source code are archived[9].

*Setting.* We implemented the one-shot algorithm, our incremental algorithm, and the generation of the JANI description of the belief support MDP into the model checker Storm [19] on top of the SMT solver z3 [38]. To compare with the one-shot algorithm for **Problem 1**, that is, for finding a policy from the initial state, we add a variant of Algorithm 3. Intuitively, any outer iteration starts with an SMT-check to see whether we find a policy covering the initial states. We realize the latter by fixing (temporarily) the $C_s$-variables. In the first iteration, this configuration and its resulting policy closely resemble the one-shot approach. For the MDP model-checking approach, we use Storm (from the C++ API) with the `dd` engine and default settings.

For the experiments, we use a MacBook Pro MV962LL/A, a single core, no randomization, and use a 6 GB memory limit. The time-out (TO) is 15 min.

[9] http://doi.org/10.5281/zenodo.4784940 or on http://github.com/sjunges/shielding-POMDPs.

*Baseline.* We compare with the one-shot algorithm including the graph-based preprocessing to identify more winning observations. We use two setups: (1) We (manually, a-priori) search for optimal hyper-parameters for each instance. We search for the smallest amount of memory possible, and for the smallest maximal rank $k$ (being a multiplicative of five) that yields a result. Guessing parameters as an "oracle" is time-consuming and unrealistic. We investigate (2) the performance of the one-shot algorithm by fixing the hyper-parameters to two memory-states and $k = 30$. These parameters provide results for most benchmarks.

*Benchmarks.* Our benchmarks involve agents operating in $N \times N$ grids, inspired by, e.g., [12,15,20,50,51]. See Fig. 2 for video stills of simulating the following benchmarks. *Rocks* is a variant of *rock sample.* The grid contains two rocks which are either valuable or dangerous to collect. To find out with certainty, the rock has to be sampled from an adjacent field. The goal is to collect a valuable rock, bring it to the drop-off zone, and not collect dangerous rocks. *Refuel* concerns a rover that shall travel from one corner to the other, while avoiding an obstacle on the diagonal. Every movement costs energy and the rover may recharge at recharging stations to its full battery capacity $E$. It receives noisy information about its position and battery level. *Evade* is a scenario where a robot needs to reach a destination and evade a faster agent. The robot has a limited range of vision ($R$), but may scan the whole grid instead of moving. A certain safe area is only accessible by the robot. *Intercept* is inverse to *Evade* in the sense that the robot aims to meet an agent before it leaves the grid via one of two available exits. On top of the view radius, the agent observes a corridor in the center of the grid. *Avoid* is a related scenario where a robot shall keep distance to patrolling agents that move with uncertain speed, yielding partial information about their position The robot may exploit their predefined routes. *Obstacle* contains static obstacles where the robot needs to reach the exit. Its initial state and movement are uncertain, and it only observes whether the current position is a trap or exit.

*Results for Challenge 1.* Table 1 details the numerical benchmark results. For each benchmark instance (columns), we report the name and relevant characteristics: the number of states ($|S|$), the number of transitions (#Tr, the edges in the graph described by the POMDP), the number of observations ($|\Omega|$), and the number of belief support states ($|b|$). For the incremental method, we provide the run time (Time, in seconds), the number of outer iterations (#Iter.) in Algorithm 3, and the number of invocations of the SMT solver (#solve), and the approximate size of the winning region ($|W|$). We then report these numbers when searching for a policy that wins from the initial state. For the one-shot method, we provide the time for the optimal parameters (on the next line)–TOs reflect settings in which we did not find any suitable parameters, and the time for the preset parameters (2,30), or N/A if no policy can be found with these parameters. Finally, for (belief-support) MDP model checking, we give only the run times.

The incremental algorithm finds winning policies for the initial state *without guessing parameters* and is often *faster* versus the one-shot approach with an

**Table 1.** Numerical results towards solving **Problem 1** and **Problem 3**.

| | | Inst. | *Rocks* $(N)$ | | *Refuel* $(N,E)$ | | *Evade* $(N,R)$ | | *Avoid* $(N,R)$ | | *Intercept* $(N,R)$ | | *Obstacle* $(N)$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 4 | 6 | 6,8 | 7,7 | 6,2 | 7,2 | 6,3 | 7,4 | 7,1 | 7,2 | 6 | 8 |
| | | $\|S\|$ | 331 | 816 | 270 | 302 | 4232 | 8108 | 5976 | 13021 | 4705 | 4705 | 37 | 65 |
| | | #Tr | 3484 | 7292 | 1301 | 1545 | 28866 | 57570 | 14373 | 33949 | 18049 | 18049 | 224 | 421 |
| | | $\|\Omega\|$ | 65 | 74 | 36 | 35 | 2202 | 4172 | 3300 | 8584 | 2002 | 2598 | 4 | 4 |
| | | $\|b\|$ | 3.5E**5** | 7.7E**25** | 5.6E**14** | 7.4E**19** | 1.1E**8** | 4.4E**11** | 1.1E**15** | 2.9E**17** | 6.4E**10** | 2.7E**9** | 1.1E**9** | 2.9E**17** |
| **incremental** | fixpoint | Time | 19 | 753 | 6 | 3 | 142 | 613 | 167 | 745 | 116 | 86 | 2 | 30 |
| | | #Iter. | 36 | 284 | 40 | 30 | 4 | 6 | 3 | 4 | 8 | 8 | 68 | 150 |
| | | #solve | 1702 | 13650 | 1023 | 528 | 681 | 1129 | 629 | 1027 | 1171 | 976 | 839 | 4291 |
| | | $\|W\|$ | 3.5E**5** | 7.7E**25** | 1.2E**11** | 2.1E**8** | 1.0E**8** | 4.2E**11** | 1.1E**15** | 2.9E**17** | 9.2E**4** | 2.9E**4** | 4.1E**7** | 3.8E**14** |
| | initial | Time | 17 | 226 | 2 | 2 | 49 | 576 | 10 | 40 | 11 | 2 | <1 | <1 |
| | | #Iter. | 29 | 65 | 2 | 4 | 1 | 1 | 1 | 1 | 2 | 1 | 10 | 12 |
| | | #solve | 1215 | 2652 | 62 | 80 | 1 | 1 | 1 | 1 | 81 | 1 | 114 | 229 |
| | | $\|W\|$ | 4.4E**4** | 1.8E**13** | 8.4E**6** | 3.7E**4** | 5.0E**7** | 1.0E**11** | 3.7E**5** | 6.9E**10** | 6.2E**3** | 2.1E**3** | 4.1E**5** | 4.5E**9** |
| **1-shot** | opt | Time | 120 | TO | 2 | <1 | 12 | 270 | 22 | 53 | 8 | 1 | 1 | 195 |
| | | Mem,k | 2,10 | ? | 2,15 | 2,15 | 1,20 | 1,30 | 1,30 | 1,25 | 2,10 | 1,10 | 6,10 | 5,50 |
| | fix | Time | TO | TO | 11 | 37 | TO | TO | TO | TO | 28 | 18 | N/A | N/A |
| **MDP** | | Time | 400 | TO | 219 | MO | TO | TO | TO | TO | TO | TO | 6 | MO |

oracle providing optimal parameters, and significantly faster than the one-shot approach with reasonably fixed parameters. In detail, *Rocks* shows that we can handle large numbers of iterations, solver invocations, and winning regions. The incremental approach scales to larger models, see e.g., *Avoid. Refuel* shows a large sensitivity of the one-shot method on the lookahead (going from 15 to 30 increases the runtime), while *Evade* shows sensitivity to memory (from 1 to 2). In contrast, the incremental approach does not rely on user-input, yet delivers comparable performance on *Refuel* or *Avoid*. It suffers slightly on *Evade*, where the one-shot approach has reduced overhead. We furthermore conclude that off-the-shelf MDP model checking is not a fast alternative. Its advantage is the guarantee to find the maximal winning region, however, for our benchmarks, maximal winning regions (empirically) coincide with the results from the incremental fixpoint approach.

*Results for Challenge 2.* Winning regions obtained from running incrementally to a fixpoint are significantly larger than when running them only until an initial winning policy is found (cf. the table), but requires extra computational effort.

If a *shielded agent* moves randomly through the grid-worlds, the larger winning regions indeed induce more permissiveness, that is, freedom to move for the agent (cf. the videos, Fig. 2). This observation can also be quantified. In Table 2, we compare the two different types of shields. For both, we give average and standard deviation over permissiveness over 250 paths. We choose to approximate permissiveness along a path as the number of cumulative actions allowed by the permissive scheduler along a path, divided by the number of cumulative actions available in the POMDP along that path. As the shield is correct by construction, each run indeed never visits avoid states and eventually reaches the target (albeit after many steps). This statement is not true for the unshielded agents.

**Table 2.** Quantification of permissiveness using fraction of allowed actions.

| Inst. | | *Rocks* ($N$) | | *Refuel* ($N$,$E$) | | *Evade* ($N$,$R$) | | *Avoid* ($N$,$R$) | | *Intercept* ($N$,$R$) | | *Obstacle* ($N$) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 4 | 6 | 6,8 | 7,7 | 6,2 | 7,2 | 6,3 | 7,4 | 7,1 | 7,2 | 6 | 8 |
| initial | avg | **0.85** | **0.81** | **0.43** | **0.36** | **0.62** | **0.50** | **0.51** | **0.56** | **0.45** | **0.47** | **0.68** | **0.74** |
| | stdev | 0.066 | 0.070 | 0.046 | 0.014 | 0.046 | 0.043 | 0.013 | 0.019 | 0.037 | 0.047 | 0.040 | 0.047 |
| fixpoint | avg | **0.88** | **0.89** | **0.77** | **0.73** | **0.86** | **0.87** | **0.78** | **0.80** | **0.78** | **0.84** | **0.73** | **0.73** |
| | stdev | 0.060 | 0.037 | 0.037 | 0.024 | 0.015 | 0.016 | 0.015 | 0.017 | 0.078 | 0.070 | 0.036 | 0.059 |

## 8 Conclusion

We provided an incremental approach to find POMDP policies that satisfy almost-sure reachability specifications. The superior scalability is demonstrated on a string of benchmarks. Furthermore, this approach allows to shield agents in POMDPs and guarantees that any exploration of an environment satisfies the specification, without needlessly restricting the freedom of the agent. We plan to investigate a tight interaction with state-of-the-art reinforcement learning and quantitative verification of POMDPs. For the latter, we expect that an explicit approach to model checking the belief-support MDP can be feasible.

## References

1. Akametalu, A.K., Kaynama, S., Fisac, J.F., Zeilinger, M.N., Gillula, J.H., Tomlin, C.J.: Reachability-based safe learning with Gaussian processes. In: CDC, pp. 1424–1431. IEEE (2014)
2. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI. AAAI Press (2018)
3. Amato, C., Bernstein, D.S., Zilberstein, S.: Optimizing fixed-size stochastic controllers for POMDPs and decentralized POMDPs. Auton. Agents Multi Agent Syst. **21**(3), 293–320 (2010). https://doi.org/10.1007/s10458-009-9103-z
4. Baier, C., Größer, M., Bertrand, N.: Probabilistic $\omega$-automata. J. ACM **59**(1), 1:1–1:52 (2012)
5. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
6. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, pp. 825–885. IOS Press (2009)
7. Bertoli, P., Cimatti, A., Pistore, M.: Towards strong cyclic planning under partial observability. In: ICAPS, pp. 354–357. AAAI (2006)
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press (2009)
9. Bloem, R., Jensen, P.G., Könighofer, B., Larsen, K.G., Lorber, F., Palmisano, A.: It's time to play safe: Shield synthesis for timed systems. CoRR abs/2006.16688 (2020)
10. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: runtime enforcement for reactive systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 533–548. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_51

11. Bork, A., Junges, S., Katoen, J.-P., Quatmann, T.: Verification of indefinite-horizon POMDPs. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 288–304. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_16
12. Brockman, G., et al.: Open AI Gym. CoRR abs/1606.01540 (2016)
13. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 151–168. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_9
14. Burns, B., Brock, O.: Sampling-based motion planning with sensing uncertainty. In: ICRA, pp. 3313–3318. IEEE (2007)
15. Chatterjee, K., Chmelik, M., Davies, J.: A symbolic SAT-based algorithm for almost-sure reachability with small strategies in POMDPs. In: AAAI, pp. 3225–3232. AAAI Press (2016)
16. Chatterjee, K., Chmelik, M., Gupta, R., Kanodia, A.: Qualitative analysis of POMDPs with temporal logic specifications for robotics applications. In: ICRA, pp. 325–330. IEEE (2015)
17. Chatterjee, K., Chmelik, M., Gupta, R., Kanodia, A.: Optimal cost almost-sure reachability in POMDPs. Artif. Intell. **234**, 26–48 (2016)
18. Chatterjee, K., Doyen, L., Henzinger, T.A.: Qualitative analysis of partially-observable Markov decision processes. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 258–269. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15155-2_24
19. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A Storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
20. Dietterich, T.G.: The MAXQ method for hierarchical reinforcement learning. In: ICML, pp. 118–126. Morgan Kaufmann (1998)
21. Dräger, K., Forejt, V., Kwiatkowska, M., Parker, D., Ujma, M.: Permissive controller synthesis for probabilistic systems. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 531–546. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_44
22. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. In: AAAI, pp. 6485–6492. AAAI Press (2018)
23. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**, 1437–1480 (2015)
24. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-regular objectives in model-free reinforcement learning. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 395–412. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_27
25. Hasanbeig, M., Abate, A., Kroening, D.: Cautious reinforcement learning with logical constraints. In: AAMAS, pp. 483–491. IFAAMAS (2020)
26. Hausknecht, M.J., Stone, P.: Deep recurrent Q-learning for partially observable MDPs. In: AAAI, pp. 29–37. AAAI Press (2015)
27. Hauskrecht, M.: Value-function approximations for partially observable Markov decision processes. J. Artif. Intell. Res. **13**, 33–94 (2000)
28. Horák, K., Bosanský, B., Chatterjee, K.: Goal-HSVI: heuristic search value iteration for goal POMDPs. In: IJCAI, pp. 4764–4770. ijcai.org (2018)
29. Jaakkola, T.S., Singh, S.P., Jordan, M.I.: Reinforcement learning algorithm for partially observable Markov decision problems. In: NIPS, pp. 345–352 (1994)

30. Jansen, N., Könighofer, B., Junges, S., Serban, A., Bloem, R.: Safe reinforcement learning using probabilistic shields (invited paper). In: CONCUR. LIPIcs, vol. 171, pp. 3:1–3:16. Schloss Dagstuhl - LZI (2020)
31. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.-P.: Safety-constrained reinforcement learning for MDPs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 130–146. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_8
32. Junges, S., Jansen, N., Seshia, S.A.: Enforcing almost-sure reachability in POMDPs. CoRR abs/2007.00085 (2020)
33. Junges, S., Jansen, N., Wimmer, R., Quatmann, T., Winterer, L., Katoen, J.P., Becker, B.: Finite-state controllers of POMDPs using parameter synthesis. In: UAI, pp. 519–529. AUAI Press (2018)
34. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1–2), 99–134 (1998)
35. Littman, M.L., Cassandra, A.R., Kaelbling, L.P.: Learning policies for partially observable environments: Scaling up. In: ICML, pp. 362–370. Morgan Kaufmann (1995)
36. Madani, O., Hanks, S., Condon, A.: On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In: AAAI, pp. 541–548. AAAI Press (1999)
37. Meuleau, N., Kim, K.E., Kaelbling, L.P., Cassandra, A.R.: Solving POMDPs by searching the space of finite policies. In: UAI, pp. 417–426. Morgan Kaufmann (1999)
38. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
39. Nam, W., Alur, R.: Active learning of plans for safety and reachability goals with partial observability. IEEE Trans. Syst. Man Cybern. Part B **40**(2), 412–420 (2010)
40. Norman, G., Parker, D., Zou, X.: Verification and control of partially observable probabilistic systems. Real-Time Syst. **53**(3), 354–402 (2017). https://doi.org/10.1007/s11241-017-9269-4
41. Pandey, B., Rintanen, J.: Planning for partial observability by SAT and graph constraints. In: ICAPS, pp. 190–198. AAAI Press (2018)
42. Pecka, M., Svoboda, T.: Safe exploration techniques for reinforcement learning - an overview. In: Hodicky, J. (ed.) MESAS 2014. LNCS, vol. 8906, pp. 357–375. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13823-7_31
43. Pineau, J., Gordon, G., Thrun, S.: Point-based value iteration: an anytime algorithm for POMDPs. In: IJCAI, pp. 1025–1032. Morgan Kaufmann (2003)
44. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE CS (1977)
45. Poupart, P., Boutilier, C.: Bounded finite state controllers. In: NIPS, pp. 823–830. MIT Press (2003)
46. Puterman, M.L.: Markov Decision Processes. Wiley, Hoboken (1994)
47. Raskin, J., Chatterjee, K., Doyen, L., Henzinger, T.A.: Algorithms for omega-regular games with imperfect information. Log. Methods Comput. Sci. **3**(3) (2007)
48. Shani, G., Pineau, J., Kaplow, R.: A survey of point-based POMDP solvers. Auton. Agent. Multi-Agent Syst. **27**(1), 1–51 (2013). https://doi.org/10.1007/s10458-012-9200-2
49. Silver, D., Veness, J.: Monte-Carlo planning in large POMDPs. In: NIPS, pp. 2164–2172 (2010)
50. Smith, T., Simmons, R.: Heuristic search value iteration for POMDPs (2004)

51. Svorenová, M., et al.: Temporal logic motion planning using POMDPs with parity objectives: case study paper. In: HSCC, pp. 233–238. ACM (2015)
52. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. The MIT Press, Cambridge (2005)
53. Turchetta, M., Berkenkamp, F., Krause, A.: Safe exploration for interactive machine learning. In: NeurIPS, pp. 2887–2897 (2019)
54. Walraven, E., Spaan, M.T.J.: Accelerated vector pruning for optimal POMDP solvers. In: AAAI, pp. 3672–3678. AAAI Press (2017)
55. Wang, Y., Chaudhuri, S., Kavraki, L.E.: Bounded policy synthesis for POMDPs with safe-reachability objectives. In: AAMAS, pp. 238–246. IFAAMAS (2018)
56. Wierstra, D., Foerster, A., Peters, J., Schmidhuber, J.: Solving deep memory POMDPs with recurrent policy gradients. In: de Sá, J.M., Alexandre, L.A., Duch, W., Mandic, D. (eds.) ICANN 2007. LNCS, vol. 4668, pp. 697–706. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74690-4_71
57. Wimmer, R., Jansen, N., Ábrahám, E., Katoen, J.P., Becker, B.: Minimal counterexamples for linear-time probabilistic verification. Theor. Comput. Sci. **549**, 61–100 (2014)
58. Winterer, L., Wimmer, R., Jansen, N., Becker, B.: Strengthening deterministic policies for POMDPs. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 115–132. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_7

# Rigorous Roundoff Error Analysis of Probabilistic Floating-Point Computations

George Constantinides[1], Fredrik Dahlqvist[1,2], Zvonimir Rakamarić[3], and Rocco Salvia[3](✉)

[1] Imperial College London, London, UK
g.constantinides@ic.ac.uk
[2] University College London, London, UK
f.dahlqvist@ucl.ac.uk
[3] University of Utah, Salt Lake City, USA
{zvonimir,rocco}@cs.utah.edu

**Abstract.** We present a detailed study of roundoff errors in probabilistic floating-point computations. We derive closed-form expressions for the distribution of roundoff errors associated with a random variable, and we prove that roundoff errors are generally close to being uncorrelated with their generating distribution. Based on these theoretical advances, we propose a model of IEEE floating-point arithmetic for numerical expressions with probabilistic inputs and an algorithm for evaluating this model. Our algorithm provides rigorous bounds to the output and error distributions of arithmetic expressions over random variables, evaluated in the presence of roundoff errors. It keeps track of complex dependencies between random variables using an SMT solver, and is capable of providing sound but tight probabilistic bounds to roundoff errors using symbolic affine arithmetic. We implemented the algorithm in the PAF tool, and evaluated it on FPBench, a standard benchmark suite for the analysis of roundoff errors. Our evaluation shows that PAF computes tighter bounds than current state-of-the-art on almost all benchmarks.

## 1 Introduction

There are two common sources of randomness in a numerical computation (a straight-line program). First, the computation might be using inherently noisy data, for example from analog sensors in cyber-physical systems such as robots, autonomous vehicles, and drones. A prime example is data from GPS sensors, whose error distribution can be described very precisely [2] and which we study in some detail in Sect. 2. Second, the computation itself might sample from random number generators. Such probabilistic numerical routines, known as Monte-Carlo methods, are used in a wide variety of tasks, such as integration [34,42], optimization [43], finance [25], fluid dynamics [32], and computer graphics [30]. We

Supported in part by the National Science Foundation awards CCF 1552975, 1704715, the Engineering and Physical Sciences Research Council (EP/P010040/1), and the Leverhulme Project Grant "Verification of Machine Learning Algorithms".

© The Author(s) 2021
A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 626–650, 2021.
https://doi.org/10.1007/978-3-030-81688-9_29

call numerical computations whose input values are sampled from some probability distributions *probabilistic computations*.

Probabilistic computations are typically implemented using floating-point arithmetic, which leads to roundoff errors being introduced in the computation. To strike the right balance between the performance and energy consumption versus the quality of the computed result, expert programmers rely on either a manual or automated floating-point error analysis to guide their design decisions. However, the current state-of-the-art approaches in this space have primary focused on *worst-case* roundoff error analysis of *deterministic* computations. So what can we say about floating-point roundoff errors in a probabilistic context? Is it possible to probabilistically quantify them by computing confidence intervals? Can we, for example, say with 99% confidence that the roundoff error of the computed result is smaller than some chosen constant? What is the distribution of outputs when roundoff errors are taken into account? In this paper, we explore these and similar questions. To answer them, we propose a rigorous – that is to say *sound* – approach to quantifying roundoff errors in probabilistic computations. Based on this approach, we develop an automatic tool that efficiently computes an overapproximate probabilistic profile of roundoff errors.

As an example, consider the floating-point arithmetic expression $(X+Y) \div Y$, where $X$ and $Y$ are random inputs represented by independent random variables. In Sect. 4, we first show how the computation in *finite-precision* of a single arithmetic operation such as $X + Y$ can be modeled as $(X + Y)(1 + \varepsilon)$, where $\varepsilon$ is also a random variable. We then show how this random variable can be computed from first principles and why it makes sense to view $(X + Y)$ and $(1 + \varepsilon)$ as independent expressions, which in turn allows us to easily compute the distribution of $(X + Y)(1 + \varepsilon)$. The distribution of $\varepsilon$ depends on that of $X + Y$, and we therefore need to evaluate arithmetic operations between random variables. When the operands are independent – as in $X + Y$ – this is standard [48], but when the operands are dependent – as in the case of the division in $(X + Y) \div Y$ – this is a hard problem. To solve it, we adopt and improve a technique for soundly bounding these distributions described in [3]. Our improvement comes from the use of an SMT solver to reason about the dependency between $(X + Y)$ and $Y$ and remove regions of the state-space with zero probability. We describe this in Sect. 6.

We can thus soundly bound the output distribution of any probabilistic computation, such as $(X+Y) \div Y$, performed in floating-point arithmetic. This gives us the ability to perform *probabilistic range analysis* and prove rigorous assertions like: 99% of the outputs of a floating-point computation are smaller than a given constant bound. In order to perform *probabilistic roundoff error analysis* we develop *symbolic affine arithmetic* in Sect. 5. This technique is combined with probabilistic range analysis to compute *conditional roundoff errors*. Specifically, we over-approximate the maximal error conditioned on the output landing in the 99% range computed by the probabilistic range analysis, meaning conditioned on the computations not returning an outlier.

We implemented our model and algorithms in a tool called PAF (for Probabilistic Analysis of Floating-point errors). We evaluated PAF on the standard floating-point benchmark suite FPBench [11], and compared its range and error analysis with the worst-case roundoff error analyzer FPTaylor [46,47] and the probabilistic roundoff error analyzer PrAn [36]. We present the results in Sect. 7, and show that FPTaylor's worst-case analysis is often overly pessimistic in the probabilistic setting, while PAF also generates tighter probabilistic error bounds than PrAn on almost all benchmarks.

We summarize our contributions as follows:

(i) We derive a closed-form expression (6) for the distribution of roundoff errors associated with a random variable. We prove that roundoff errors are generally close to being uncorrelated with their input distribution.
(ii) Based on these results we propose a model of IEEE 754 floating-point arithmetic for numerical expressions with probabilistic inputs.
(iii) We evaluate this model by developing a new algorithm for rigorously bounding the output range and roundoff error distributions of floating-point arithmetic expressions with probabilistic inputs.
(iv) We implement this model in the PAF tool,[1] and perform probabilistic range and roundoff error analysis on a standard benchmark suite. Our comparison with the current state-of-the-art shows the advantages of our approach in terms of computing tighter, and yet still rigorous, probabilistic bounds.

## 2 Motivating Example

GPS sensors are inherently noisy. Bornholt [1] shows that the conditional probability of the true coordinates given a GPS reading is distributed according to a Rayleigh distribution. Interestingly, since the density of any Rayleigh distribution is always zero at $x = 0$, it is extremely unlikely that the true coordinates lie in a small neighborhood of those given by the GPS reading. This leads to errors, and hence the sensed coordinates should be corrected by adding a probabilistic error term which, on average, shifts the observed coordinates into an area of high probability for the true coordinates [1,2]. The latitude correction is given by:

$$\texttt{TrueLat} = \texttt{GPSLat} + ((\texttt{radius} * \texttt{sin}(\texttt{angle})) * \texttt{DPERM}), \tag{1}$$

where `radius` is Rayleigh distributed, `angle` uniformly distributed, `GPSLat` is the latitude, and `DPERM` a constant for converting meters into degrees.

A developer trying to strike the right balance between resources, such as energy consumption or execution time, versus the accuracy of the computation, might want to run a rigorous worst-case floating-point analysis tool to determine which floating-point format is accurate enough to process GPS signals. This is mandatory if the developer requires rigorous error bounds holding with 100% certainty. The problem when analyzing a piece of code involving (1) is that the Rayleigh distribution has $[0, \infty)$ as its support, and *any* worst-case roundoff error analysis will return an infinite error bound in this situation. To get a meaningful (numeric) error bound, we need to truncate the support of the distribution. The most conservative truncation is $[0, max]$, where $max$ is the largest representable number (not causing an overflow) at the target floating-point precision format.

[1] PAF is open source and publicly available at https://github.com/soarlab/paf.

**Table 1.** Roundoff error analysis for the probabilistic latitude correction of (1).

| Precision | Max | FPTaylor | PAF 100% | PAF 99.9999% | |
|---|---|---|---|---|---|
| | | | | Absolute | Meters |
| Double | $\approx 10^{307}$ | 4.3e+286 | 4.3e+286 | 4.1e−15 | 4.5e−10 |
| Single | $\approx 10^{38}$ | 2.1e+26 | 2.1e+26 | 3.7e−06 | 4.1e−1 |
| Half | $\approx 10^{4}$ | 2.5e−2 | 2.5e−2 | 2.4e−2 | 2667 |

In Table 1, we report a detailed roundoff error analysis of (1) implemented in IEEE 754 double-, single-, and half-precision formats, with `GPSLat` set to the latitude of the Greenwich observatory. With each floating-point format, we associate the range $[0, max]$ of the truncated Rayleigh distribution. We compute worst-case roundoff error bounds for (1) with the state-of-the-art error analyzer FPTaylor [47] and with our tool PAF by setting the confidence interval to 100%. As expected, the error bounds from the two tools are identical. Finally, we compute the 99.9999% *conditional roundoff error* using PAF. This value is an upper bound to the roundoff error *conditioned* on the computation having landed in an interval capturing 99.9999% of all possible outputs. Column Absolute gives the error in degrees and Meters in meters ($1° \approx$111km).

By looking at the results obtained without our *probabilistic error analysis* (columns FPTaylor and PAF 100%), the developer might *erroneously* conclude that half-precision format is the most appropriate to implement (1) because it results in the smallest error bound. However, with the information provided by the 99.9999% *conditional roundoff error*, the developer can see that the *average* error is many orders of magnitude smaller than the worst-case scenarios. Armed with this information, the developer can conclude that with a roundoff error of roughly 40 cm (4.1e−1 ms) when correcting 99.9999% of GPS latitude readings, working in single-precision is an adequate compromise between efficiency and accuracy of the computation.

This motivates the innovative concept of *probabilistic precision tuning*, evolved from standard worst-case precision tuning [5,12], to determine which floating-point format is the most appropriate for a given computation. As an example, let us do a probabilistic precision tuning exercise for the latitude correction computation of (1). We truncate the Rayleigh distribution in the interval $[0, 10^{307}]$, and assume we can tolerate up to 1e−5 roundoff error (roughly 1 m). First, we manually perform worst-case precision tuning using FPTaylor to determine that the minimal floating-point format not violating the given error bound needs 1022 mantissa and 11 exponent bits. Such large custom format is prohibitively expensive, in particular for devices performing frequent GPS readings, like smartphones or smartwatches. Conversely, when we manually perform probabilistic precision tuning using PAF with a confidence interval set to 99.9999%, we determine we need only 22 mantissa and 11 exponent bits. Thanks to PAF, the developer can provide a custom confidence interval of interest to the probabilistic precision tuning routine to adjust for the extremely unlikely corner cases like the ones we described for (1), and ultimately obtain more optimal tuning results.

## 3 Preliminaries

### 3.1 Floating-Point Arithmetic

Given a *precision* $p \in \mathbb{N}$ and an *exponent range* $[e_{min}, e_{max}] \triangleq \{n \mid n \in \mathbb{N} \wedge e_{min} \leq n \leq e_{max}\}$, we define $\mathbb{F}(p, e_{min}, e_{max})$, or simply $\mathbb{F}$ if there is no ambiguity, as the set of extended real numbers

$$\mathbb{F} \triangleq \left\{ (-1)^s 2^e \left(1 + \frac{k}{2^p}\right) \middle| s \in \{0,1\}, e \in [e_{min}, e_{max}], 0 \leq k < 2^p \right\} \cup \{-\infty, 0, \infty\}$$

Elements $z = z(s, e, k) \in \mathbb{F}$ will be called *floating-point representable numbers* (for the given precision $p$ and exponent range $[e_{min}, e_{max}]$) and we will use the variable $z$ to represent them. The variable $s$ will be called the *sign*, the variable $e$ the *exponent*, and the variable $k$ the *significand* of $z(s, e, k)$.

Next, we introduce a *rounding map* $\text{Round} : \mathbb{R} \to \mathbb{F}$ that rounds to nearest (or to $-\infty/\infty$ for values smaller/greater than the smallest/largest finite element of $\mathbb{F}$) and follows any of the IEEE 754 rounding modes in case of a tie. We will not worry about which choice is made since the set of mid-points will always have probability zero for the distributions we will be working with. All choices are thus equivalent, probabilistically speaking, and what happens in a tie can therefore be left unspecified. We will denote the extended real line by $\overline{\mathbb{R}} \triangleq \mathbb{R} \cup \{-\infty, \infty\}$. The (signed) *absolute error function* $\text{err}_{\text{abs}} : \mathbb{R} \to \overline{\mathbb{R}}$ is defined as: $\text{err}_{\text{abs}}(x) = x - \text{Round}(x)$. We define the sets $\lfloor z, z \rceil \triangleq \{y \in \mathbb{R} \mid \text{Round}(y) = \text{Round}(z)\}$. Thus if $z \in \mathbb{F}$, then $\lfloor z, z \rceil$ is the collection of all reals rounding to $z$. As the reader will see, the basic result of Sect. 4 (Eq. (5)) is expressed entirely using the notation $\lfloor z, z \rceil$ which is parametric in the choice of the Round function. It follows that our results apply to rounding modes other that round-to-nearest with minimal changes. The *relative error function* $\text{err}_{\text{rel}} : \mathbb{R} \setminus \{0\} \to \overline{\mathbb{R}}$ is defined by

$$\text{err}_{\text{rel}}(x) = \frac{x - \text{Round}(x)}{x}.$$

Note that $\text{err}_{\text{rel}}(x) = 1$ on $\lfloor 0, 0 \rceil \setminus \{0\}$, $\text{err}_{\text{rel}}(x) = \infty$ on $\lfloor -\infty, -\infty \rceil$ and $\text{err}_{\text{rel}}(x) = -\infty$ on $\lfloor \infty, \infty \rceil$. Recall also the fact [26] that $-2^{-(p+1)} < \text{err}_{\text{rel}}(x) < 2^{-(p+1)}$ outside of $\lfloor 0, 0 \rceil \cup \lfloor -\infty, -\infty \rceil \cup \lfloor \infty, \infty \rceil$. The quantity $2^{-(p+1)}$ is usually called the *unit roundoff* and will be denoted by $u$.

For $z_1, z_2 \in \mathbb{F}$ and $\text{op} \in \{+, -, \times, \div\}$ an (infinite-precision) arithmetic operation, the traditional model of IEEE 754 floating-point arithmetic [26,39] states that the finite-precision implementation $\mathtt{op_m}$ of op must satisfy

$$z_1 \ \mathtt{op_m} \ z_2 = (z_1 \ \text{op} \ z_2)(1 + \delta) \qquad |\delta| \leq u, \tag{2}$$

We leave dealing with subnormal floating-point numbers to future work. The model given by Eq. (2) stipulates that the implementation of an arithmetic operation can induce a relative error of magnitude *at most* $u$. The exact size of the error is, however, not specified and Eq. (2) is therefore a *non-deterministic* *model of computation*. It follows that numerical analyses based on Eq. (2) must consider *all* possible relative errors $\delta$ and are fundamentally *worst-case* analyses. Since the output of such a program might be the input of another, one should also consider non-deterministic inputs, and this is indeed what happens with automated tools for roundoff error analysis, such as Daisy [12] or FPTaylor [46, 47], which require for each variable of the program a (bounded) range of possible values in order to perform a worst-case analysis (*cf.* GPS example in Sect. 2).

In this paper, we study a model formally similar to Eq. (2), namely

$$z_1 \ \mathtt{op_m}\ z_2 = (z_1 \text{ op } z_2)(1+\delta) \qquad \delta \sim \mathit{dist}. \tag{3}$$

The difference is that $\delta$ is now *distributed according to dist*, a probability distribution whose support is $[-u, u]$. In other words, we move from a non-deterministic to a *probabilistic* model of roundoff errors. This is similar to the 'Monte Carlo arithmetic' of [41], but whilst *op. cit. postulates* that *dist* is the uniform distribution on $[-u, u]$, we compute *dist* from first principles in Sect. 4.

## 3.2 Probability Theory

To fix the notation and be self-contained, we present some basic notions of probability theory which are essential to what follows.

**Cumulative Distribution Functions and Probability Density Functions.** We assume that the reader is (at least intuitively) familiar with the notion of a (real) random variable. Given a random variable $X$ we define its Cumulative Distribution Function (CDF) as the function $c(t) \triangleq \mathbb{P}[X \leq t]$. If there exists a non-negative integrable function $d : \mathbb{R} \to \mathbb{R}$ such that

$$c(t) \triangleq \mathbb{P}[X \leq t] = \int_{-\infty}^{t} d(t)\ dt$$

then we call $d(t)$ the Probability Density Function (PDF) of $X$. If it exists, then it can be recovered from the CDF by differentiation $d(t) = \frac{\partial}{\partial t} c(t)$ by the fundamental theorem of calculus.

Not all random variables have a PDF: consider the random variable which takes value 0 with probability $1/2$ and value 1 with probability $1/2$. For this random variable it is impossible to write $\mathbb{P}[X \leq t] = \int d(t)\ dt$. Instead, we will write the distribution of such a variable using the so-called Dirac delta measure at 0 and 1 as $1/2\delta_0 + 1/2\delta_1$. It is possible for a random variable to have a PDF covering part of its distribution – its *continuous part* – and a sum of Dirac deltas covering the rest of its distribution – its *discrete part*. We will encounter examples of such random variables in Sect. 4. Finally, if $X$ is a random variable and $f : \mathbb{R} \to \mathbb{R}$ is a measurable function, then $f(X)$ is a random variable. In particular $\mathrm{err}_{\mathrm{rel}}(X)$ is a random variable which we will describe in Sect. 4.

**Arithmetic on Random Variables.** Suppose $X, Y$ are *independent* random variables with PDFs $f_X$ and $f_Y$, respectively. Using the arithmetic operations we can form new random variables $X + Y, X - Y, X \times Y, X \div Y$. The PDFs of these new random variables can be expressed as operations on $f_X$ and $f_Y$, which can be found in [48]. It is important to note that these operations are only valid if $X$ and $Y$ are assumed to be independent. When an arithmetic expression containing variable repetitions is given a random variable interpretation, this independence can no longer be assumed. In the expression $(X + Y) \div Y$ the sub-term $(X + Y)$ can be interpreted by the formulas of [48] if $X, Y$ are independent. However, the sub-terms $X + Y$ and $Y$ cannot be interpreted in this way since $X + Y$ and $Y$ are clearly not independent random variables.

**Soundly Bounding Probabilities.** The constraint that the distribution of a random variable must integrate to 1 makes it impossible to order random variables in the 'natural' way: if $\mathbb{P}[X \in A] \leq \mathbb{P}[Y \in A]$, then $\mathbb{P}[Y \in A^c] \leq \mathbb{P}[X \in A^c]$, i.e., we cannot say that $X \leq Y$ if $\mathbb{P}[X \in A] \leq \mathbb{P}[Y \in A]$. This means that we cannot quantify our probabilistic uncertainty about a random variable by sandwiching it between two other random variables as one would do with reals or real-valued functions. One solution is to restrict the sets used in the comparison, i.e., declare that $X \leq Y$ iff $\mathbb{P}[X \in A] \leq \mathbb{P}[Y \in A]$ for $A$ ranging over a given set of 'test subsets'. Such an order can be defined by taking as 'test subsets' the intervals $(-\infty, x]$ [44]. This order is called the *stochastic order*. It follows from the definition of the CDF that this order can be defined by simply saying that $X \leq Y$ iff $c_X \leq c_Y$, where $c_X$ and $c_Y$ are the CDFs of $X$ and $Y$, respectively. If it is possible to sandwich an unknown random variable $X$ between known lower and upper bounds $X_{lower} \leq X \leq X_{upper}$ using the stochastic order then it becomes possible to give sound bounds to the quantities $\mathbb{P}[X \in [a, b]]$ via

$$\mathbb{P}[X \in [a, b]] = c_X(b) - c_X(a) \leq c_{X_{upper}}(b) - c_{X_{lower}}(a)$$

**P-Boxes and DS-Structures.** As mentioned above, giving a random variable interpretation to an arithmetic expression containing variable repetitions cannot be done using the arithmetic of [48]. In fact, these interpretations are in general analytically intractable. Hence, a common approach is to give up on soundness and approximate such distributions using Monte-Carlo simulations. We use this approach in our experiments to assess the quality of our sound results. However, we will also provide sound under- and over-approximations of the distribution of arithmetic expressions over random variables using the stochastic order discussed above. Since $X_{lower} \leq X \leq X_{upper}$ is equivalent to saying that $c_{X_{lower}}(x) \leq c_X(x) \leq c_{X_{upper}}(x)$, the fundamental approximating structure will be a pair of CDFs satisfying $c_1(x) \leq c_2(x)$. Such a structure is known in the literature as a *p-box* [19], and has already been used in the context of probabilistic roundoff errors in related work [3,36]. The data of a p-box is equivalent to a pair of sandwiching distributions for the stochastic order.

A *Dempster-Shafer structure* (DS-structure) of size $N$ is a collection (i.e., set) of interval-probability pairs $\{([x_0, y_0], p_0), ([x_1, y_2], p_1), .., ([x_N, y_N], p_N)\}$ where $\sum_{i=0}^{N} p_i = 1$. The intervals in the collection might overlap. One can always convert a DS-structure to a p-box and back again [19], but arithmetic operations are much easier to perform on DS-structures than on p-boxes ([3]), which is why we will use DS-structures in the algorithm described in Sect. 6.

## 4 Distribution of Floating-Point Roundoff Errors

Our tool PAF computes *probabilistic* roundoff errors by conditioning the maximization of symbolic affine form (presented in Sect. 5) on the output of the computation landing in a confidence interval. The purpose of this section is to provide the necessary probabilistic tools to compute these intervals. In other words, this section provides the foundations of *probabilistic range analysis*. All proofs can be found in the extended version [7].

### 4.1 Derivation of the Distribution of Rounding Errors

Recall the probabilistic model of Eq. (3) where op is an infinite-precision arithmetic operation and $\mathtt{op_m}$ its finite-precision implementation:

$$z_1 \ \mathtt{op_m}\ z_2 = (z_1 \text{ op } z_2)(1+\delta) \qquad \delta \sim \mathit{dist}.$$

Let us also assume that $z_1, z_2$ are random variables with known distributions. Then $z_1$ op $z_2$ is also a random variable which can (in principle) be computed. Since the IEEE 754 standard states that $z_1 \ \mathtt{op_m}\ z_2$ is computed by rounding the infinite precision operation $z_1$ op $z_2$, it is a completely natural consequence of the standard to require that $\delta$ is simply be given by

$$\delta = \mathrm{err}_{\mathrm{rel}}(z_1 \text{ op } z_2)$$

Thus, *dist* is the distribution of the random variable $\mathrm{err}_{\mathrm{rel}}(z_1 \text{ op } z_2)$. More generally, if $X$ is a random variable with know distribution, we will show how to compute the distribution *dist* of the random variable

$$\mathrm{err}_{\mathrm{rel}}(X) = \frac{X - \mathrm{Round}(X)}{X}.$$

We choose to express the distribution *dist* of relative errors *in multiples of the unit roundoff* $u$. This choice is arbitrary, but it allows us to work with a distribution on the conceptually and numerically convenient interval $[-1, 1]$, since the absolute value of the relative error is strictly bounded by $u$ (see Sect. 3.1), rather than the interval $[-u, u]$.

To compute the density function of *dist*, we proceed as described in Sect. 3.2 by first computing the CDF $c(t)$ and then taking its derivative. Recall first from Sect. 3.1 that $\mathrm{err}_{\mathrm{rel}}(x) = 1$ if $x \in \lfloor 0, 0\rceil \setminus \{0\}$, $\mathrm{err}_{\mathrm{rel}}(x) = \infty$ if $x \in \lfloor -\infty, -\infty\rceil$, $\mathrm{err}_{\mathrm{rel}}(x) = -\infty$ if $x \in \lfloor \infty, \infty\rceil$, and $-u \le \mathrm{err}_{\mathrm{rel}}(x) \le u$ elsewhere. Thus:

$$\mathbb{P}\left[\mathrm{err}_{\mathrm{rel}}(X) = -\infty\right] = \mathbb{P}\left[X \in \lfloor \infty, \infty\rceil\right] \qquad \mathbb{P}\left[\mathrm{err}_{\mathrm{rel}}(X) = 1\right] = \mathbb{P}\left[X \in \lfloor 0, 0\rceil\right]$$
$$\mathbb{P}\left[\mathrm{err}_{\mathrm{rel}}(X) = \infty\right] = \mathbb{P}\left[X \in \lfloor -\infty, -\infty\rceil\right]$$

In other words, the probability measure corresponding to $\mathrm{err}_{\mathrm{rel}}$ has three discrete components at $\{-\infty\}, \{1\}$, and $\{\infty\}$, which cannot be accounted for by a PDF (see Sect. 3.2). It follows that the probability measure *dist* is given by

$$\mathit{dist}_c + \mathbb{P}\left[X \in \lfloor 0, 0\rceil\right]\delta_1 + \mathbb{P}\left[X \in \lfloor -\infty, -\infty\rceil\right]\delta_\infty + \mathbb{P}\left[X \in \lfloor \infty, \infty\rceil\right]\delta_{-\infty} \quad (4)$$

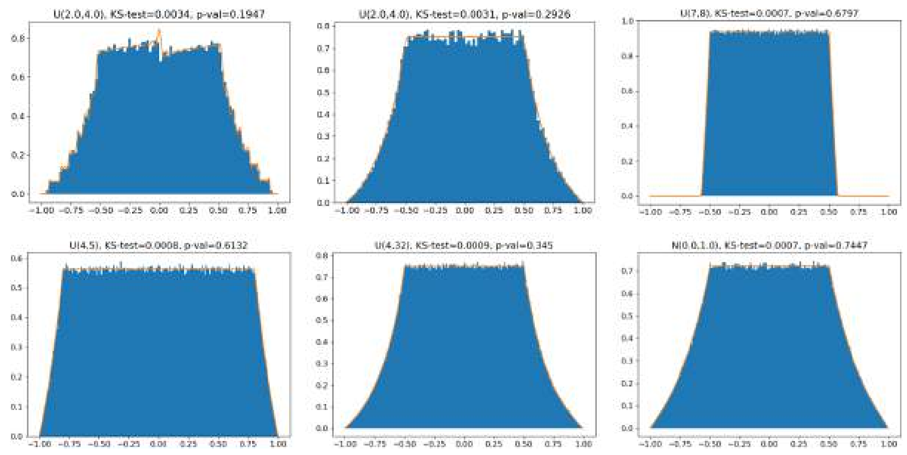


**Fig. 1.** Theoretical vs. empirical error distribution, clockwise from top-left: (i) Eq. (5) for $\mathsf{Unif}(2,4)$ 3 bit exponent, 4 bit significand, (ii) Eq. (5) for $\mathsf{Unif}(2,4)$ in half-precision, (iii) Eq. (6) for $\mathsf{Unif}(7,8)$ in single-precision, (iv) Eq. (6) for $\mathsf{Unif}(4,5)$ in single-precision, (v) Eq. (6) for $\mathsf{Unif}(4,32)$ in single-precision, (vi) Eq. (6) for $\mathsf{Norm}(0,1)$ in single-precision.

where $dist_c$ is a continuous measure that is not quite a probability measure since its total mass is $1 - \mathbb{P}[X \in \lfloor 0,0 \rceil] - \mathbb{P}[X \in \lfloor -\infty, -\infty \rceil] - \mathbb{P}[X \in \lfloor \infty, \infty \rceil]$. In general, $dist_c$ integrates to 1 in machine precision since $\mathbb{P}[X \in \lfloor 0,0 \rceil]$ is of the order of the smallest positive floating-point representable number, and the PDF of $X$ rounds to 0 way before it reaches the smallest/largest floating-point representable number. However in order to be sound, we must in general include these three discrete components to our computations. The density $dist_c$ is given explicitly by the following result whose proof can already be found in [9].

**Theorem 1.** *Let $X$ be a real random variable with PDF $f$. The continuous part $dist_c$ of the distribution of $\mathrm{err}_{\mathrm{rel}}(X)$ has a PDF given by*

$$d(t) = \sum_{z \in \mathbb{F} \setminus \{-\infty, 0, \infty\}} \mathbb{1}_{\lfloor z,z \rceil}\left(\frac{z}{1-tu}\right) f\left(\frac{z}{1-tu}\right) \frac{u\,|z|}{(1-tu)^2}, \tag{5}$$

*where $\mathbb{1}_A(x)$ is the indicator function which returns 1 if $x \in A$ and 0 otherwise.*

Figure 1 (i) and (ii) shows an implementation of Eq. (5) applied to the distribution $\mathsf{Unif}(2,4)$, first in very low precision (3 bit exponent, 4 bit significand) and then in half-precision. The theoretical density is plotted alongside a histogram of the relative error incurred when rounding 100,000 samples to low precision (computed in double-precision). The reported statistic is the K-S (Kolmogorov-Smirnov) test which measures the likelihood that a collection of samples were drawn from a given distribution. This test reports that we cannot reject the hypothesis that the samples are drawn from the corresponding density. Note how in low precision the term in $\frac{1}{(1-tu)^2}$ induces a visible asymmetry on the central section of the distribution. This effect is much less pronounced in half-precision.

For low precisions, say up to half-precision, it is computationally feasible to explicitly go through all floating-point numbers and compute the density of the roundoff error distribution *dist* directly from Eq. (5). However, this rapidly becomes prohibitively computationally expensive for higher precisions (since the number of floating-point representable numbers grows exponentially).

## 4.2 High-Precision Case

As the working precision increases, a regime changes occurs: on the one hand it becomes practically impossible to enumerate all floating-point representable numbers as done in Eq. (5), but on the other hand sufficiently well-behaved density functions are numerically close to being constant at the scale of an interval between two floating-point representable numbers. We exploit this smoothness to overcome the combinatorial limit imposed by Eq. (5).

**Theorem 2.** *Let $X$ be a real random variable with PDF $f$. The continuous part $dist_c$ of the distribution of $\mathrm{err}_{\mathrm{rel}}(X)$ has a PDF given by $d_c(t) = d_{hp}(t) + R(t)$ where $d_{hp}(t)$ is the function on $[-1, 1]$ defined by*

$$d_{hp}(t) = \begin{cases} \frac{1}{1-tu} \sum\limits_{s,e=e_{min}+1}^{e_{max}-1} \int_{(-1)^s 2^e(1-u)}^{(-1)^s 2^e(2-u)} \frac{|x|}{2^{e+1}} f(x)\ dx & |t| \leq \frac{1}{2} \\ \\ \frac{1}{1-tu} \sum\limits_{s,e=e_{min}+1}^{e_{max}-1} \int_{(-1)^s 2^e(1-u)}^{(-1)^s 2^e(\frac{1}{|t|}-u)} \frac{|x|}{2^{e+1}} f(x)\ dx & \frac{1}{2} < |t| \leq 1 \end{cases} \tag{6}$$

*and $R(t)$ is an error whose total contribution $|R| \triangleq \int_{-1}^{1} |R(t)| dt$ can be bounded by*

$$|R| \leq \mathbb{P}[\mathrm{Round}(X) = z(s, e_{min}, k)] + \mathbb{P}[\mathrm{Round}(X) = z(s, e_{max}, k)] + \frac{3}{4}\left(\sum_{s, e_{min}<e<e_{max}} |f'(\xi_{e,s})\xi_{e,s} + f(\xi_{e,s})| \frac{2^{2e}}{2^p}\right)$$

*where for each exponent $e$ and sign $s$, $\xi_{e,s}$ is a point in $[z(s, e, 0), z(s, e, 2^p - 1)]$ if $s = 0$ and in $[z(s, e, 2^p - 1), z(s, e, 0)]$ if $s = 1$.*

Note how Eq. (6) reduces the sum over *all* floating-point representable numbers in Eq. (5) to a sum over *the exponents* by exploiting the regularity of $f$. Note also that since $f$ is a PDF, it usually decreases very quickly away from 0, and its derivative decreases even quicker and $|R|$ thus tends to be very small and $|R| \to 0$ as the precision $p \to \infty$.

Figure 1 shows Eq. (6) for: (i) the distribution $\mathsf{Unif}(7, 8)$ where large significands are more likely, (ii) the distribution $\mathsf{Unif}(4, 5)$ where small significands are more likely, (iii) the distribution $\mathsf{Unif}(4, 32)$ where significands are equally likely, and (iv) the distribution $\mathsf{Norm}(0,1)$ with infinite support. The graphs show the density function given by Eq. (6) in single-precision versus a histogram of the relative error incurred when rounding 1,000,000 samples to single-precision (computed in double-precision). The K-S test reports that we cannot reject the hypothesis that the samples are drawn from the corresponding distributions.

## 4.3 Typical Distribution

The distributions depicted in graphs (ii), (v) and (vi) of Fig. 1 are very similar, despite being computed from very different input distributions. What they have in common is that their input distributions have the property that all significands in their supports are equally likely. We show that under this assumption, the distribution of roundoff errors given by Eq. (5) converges to a unique density as the precision increases, irrespective of the input distribution! Since significands are frequently equiprobable (it is the case for a third of our benchmarks), this density is of great practical importance. If one had to choose 'the' canonical distribution for roundoff errors, we claim that the density given below should be this distribution, and we therefore call it the *typical distribution*; we depict it in Fig. 2 and formalize it with the following theorem, which can mostly be found in [9].

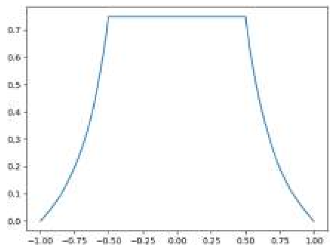


**Fig. 2.** Typical distribution.

**Theorem 3.** *If $X$ is a random variable such that $\mathbb{P}\left[\text{Round}(X) = z(s, e, k_0)\right] = \frac{1}{2^p}$ for any significand $k_0$, then*

$$d_{typ}(t) \triangleq \lim_{p\to\infty} d(t) = \begin{cases} \frac{3}{4} & |t| \leq \frac{1}{2} \\ \frac{1}{2}\left(\frac{1}{t} - 1\right) + \frac{1}{4}\left(\frac{1}{t} - 1\right)^2 & |t| > \frac{1}{2} \end{cases} \tag{7}$$

*where $d(t)$ is the exact density given by Eq.* (5).

## 4.4 Covariance Structure

The result above can be interpreted as saying that if $X$ is such that all mantissas are equiprobable, then $X$ and $\text{err}_{\text{rel}}(X)$ are asymptotically independent (as $p \to \infty$). Much more generally, we now show that if a random variable $X$ has a sufficiently regular PDF, it is close to being uncorrelated from $\text{err}_{\text{rel}}(X)$. Formally, we prove that the covariance

$$\text{Cov}(X, \text{err}_{\text{rel}}(X)) = \mathbb{E}\left[X.\text{err}_{\text{rel}}(X)\right] - \mathbb{E}\left[X\right]\mathbb{E}\left[\text{err}_{\text{rel}}(X)\right] \tag{8}$$

is small, specifically of the order of $u$. Note that the expectation in the first summand above is taken w.r.t. the joint distribution of $X$ and $\text{err}_{\text{rel}}(X)$.

The main technical obstacles to proving that the expression above is small are that $\mathbb{E}\left[\text{err}_{\text{rel}}(X)\right]$ turns out to be difficult to compute (we only manage to bound it) and that the joint distribution $\mathbb{P}[X \in A \wedge \mathrm{err_{rel}}(X) \in B]$ does not have a PDF since it is not continuous w.r.t. the Lebesgue measure on $\mathbb{R}^2$. Indeed, it is supported by the graph of the function $\mathrm{err_{rel}}$ which has a Lebesgue measure of 0. This does not mean that it is impossible to compute the expectation

$$\mathbb{E}[X.\mathrm{err_{rel}}(X)] = \int_{\mathbb{R}^2} xut \; d\mathbb{P} \tag{9}$$

but it is necessary to use some more advanced probability theory. We will make the simplifying assumption that the density of $X$ is constant on each interval $\lfloor z, z \rceil$ in order to keep the proof manageable. In practice this is an extremely good approximation. Without this assumption, we would need to add an error term similar to that of Theorem 2 to the expression below. This is not conceptually difficult, but it is messy, and would distract from the main aim of the following theorem which is to bound $\mathbb{E}[\mathrm{err_{rel}}(X)]$, compute $\mathbb{E}[X.\mathrm{err_{rel}}(X)]$, and show that the covariance between $X$ and $\mathrm{err_{rel}}(X)$ is typically of the order of $u$.

**Theorem 4.** *If the density of $X$ is piecewise constant on intervals $\lfloor z, z \rceil$, then*

$$\left(L - \mathbb{E}[X]\, K\frac{u}{6}\right) \leq \mathrm{Cov}(X, \mathrm{err_{rel}}(X)) \leq \left(L - \mathbb{E}[X]\, K\frac{4u}{3}\right)$$

*where* $L = \sum_{s,e} f((-1)^s 2^e)(-1)^s 2^{2e}\frac{3u^2}{2}$ *and* $K = \sum_{s,e=e_{min}+1}^{e_{max}-1} \int_{(-1)^s 2^e(1-u)}^{(-1)^s 2^e(2-u)} \frac{|x|}{2^{e+1}}$ *$f(x)\ dx$.*

If the distribution of $X$ is centered (i.e., $\mathbb{E}[X] = 0$) then $L$ is the exact value of the covariance, and it is worth noting that $L$ is fundamentally an artifact of the floating-point representation and is due to the fact that the intervals $\lfloor 2^e, 2^e \rceil$ are not symmetric. More generally, for $\mathbb{E}[X]$ of the order of, say, 2, the covariance will be small (of the order of $u$) as $K \leq 1$ (since $|x| \leq 2^{e+1}$ in each summand). For very large values of $\mathbb{E}[X]$ it is worth noting that there is a high chance that $L$ is also be very large, partially canceling $\mathbb{E}[X]$. An illustration of this is given by the *doppler* benchmark examined in Sect. 7, an outlier as it has an input variable with range [20, 20000]. Nevertheless, even for this benchmark the bounds of Theorem 4 still give a small covariance of the order of 0.001.

## 4.5 Error Terms and P-Boxes

In low-precision we can use the exact formula Eq. (5) to compute the error distribution. However, in high-precision, approximations (typically extremely good) like Eqs. (6) and (7) must be used. In order to remain sound in the implementation of our model (see Sect. 6) we must account for the error made by this approximation. We have not got the space to discuss the error made by Eq. (7), but taking the term $|R|$ of Theorem 2 as an illustration, we can use the notion of p-box described in Sect. 3.2 to create an object which soundly approximates the error distribution. We proceed as follows: since $|R|$ bounds the total error accumulated over all $t \in [-1, 1]$, we can soundly bound the CDF $c(t)$ of the error distribution given by Eq. (6) by using the p-box

$$c^-(t) = \max(0, c(t) - |R|) \qquad \text{and} \qquad c^+(t) = \min(1, c(t) + |R|)$$

## 5 Symbolic Affine Arithmetic

In this section, we introduce *symbolic affine arithmetic*, which we employ to generate the symbolic form for the roundoff error that we use in Sect. 6.3. Affine arithmetic [6] is a model for range analysis that extends classic interval arithmetic [40] with information about linear correlations between operands. Symbolic affine arithmetic extends standard affine arithmetic by keeping the coefficients of the noise terms *symbolic*. We define a *symbolic affine form* as

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \epsilon_i, \qquad \text{where } \epsilon_i \in [-1, 1]. \tag{10}$$

We call $x_0$ the central symbol of the affine form, while $x_i$ are the symbolic coefficients for the noise terms $\epsilon_i$. We can always convert a symbolic affine form to its corresponding interval representation. This can be done using interval arithmetic or, to avoid precision loss, using a global optimizer.

Affine operations between symbolic forms follow the usual rules, such as

$$\alpha\hat{x} + \beta\hat{y} + \zeta = \alpha x_0 + \beta y_0 + \zeta + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\epsilon_i$$

Non-linear operations cannot be represented exactly using an affine form. Hence, we approximate them like in standard affine arithmetic [49].

**Sound Error Analysis with Symbolic Affine Arithmetic.** We now show how the roundoff errors get propagated through the four arithmetic operations. We apply these propagation rules to an arithmetic expression to accurately keep track of the roundoff errors. Since the (absolute) roundoff error directly depends on the range of a computation, we describe range and error together as a pair `(range: Symbol,` $\widehat{err}$`: Symbolic Affine Form)`. Here, `range` represents the infinite-precision range of the computation, while $\widehat{err}$ is the symbolic affine form for the roundoff error in floating-point precision. Unary operators (e.g., rounding) take as input a (range, error form) pair, and return a new output pair; binary operators take as input two pairs, one per operand. For linear operators, the ranges and errors get propagated using the standard rules of affine arithmetic.

For the multiplication, we distribute each term in the first operand to every term in the second operand:

$$(\mathtt{x},\ \widehat{err}_x) * (\mathtt{y},\ \widehat{err}_y) = (\mathtt{x*y},\ \mathtt{x} * \widehat{err}_y + \mathtt{y} * \widehat{err}_x + \widehat{err}_x * \widehat{err}_y)$$

The output range is the product of the input ranges and the remaining terms contribute to the error. Only the last (quadratic) expression cannot be represented exactly in symbolic affine arithmetic; we bound such non-linearities using a global optimizer. The division is computed as the term-wise multiplication of the numerator with the inverse of the denominator. Hence, we need the inverse of the denominator error form, and then we can proceed as for multiplication. To compute the inverse, we leverage the symbolic expansion used in FPTaylor [46].

Finally, after every operation we apply the unary rounding operator from Eq. (2). The infinite-precision range is not affected by rounding. The rounding operator appends a fresh noise term to the symbolic error form. The coefficient for the new noise term is the (symbolic) floating-point range given by the sum of the input range with the input error form.

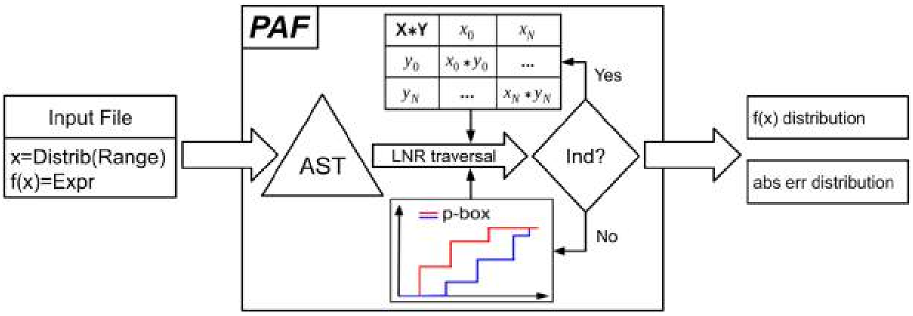


**Fig. 3.** Toolflow of PAF.

## 6 Algorithm and Implementation

In this section, we describe our probabilistic model of floating-point arithmetic and how we implement it in a prototype named PAF (for Probabilistic Analysis of Floating-point errors). Figure 3 shows the toolflow of PAF.

### 6.1 Probabilistic Model

PAF takes as input a text file describing a probabilistic floating-point computation and its input distributions. The kinds of computations we support are captured with this simple grammar:

$$\mathtt{t} ::= \mathtt{z} \mid \mathtt{x_i} \mid \mathtt{t}\ \mathtt{op_m}\ \mathtt{t} \qquad \mathtt{z} \in \mathbb{F}, \mathtt{i} \in \mathbb{N},\ \mathtt{op_m}\ \in \{+, -, \times, \div\}$$

Following [8,31], we interpret each computation $\mathtt{t}$ given by the grammar as a random variable. We define the interpretation map $[\![-]\!]$ over the computation tree inductively. The base case is given by $[\![\mathtt{z}(s,e,k)]\!] \triangleq (-1)^s 2^e(1 + k2^{-p})$ and $[\![\mathtt{x_i}]\!] \triangleq X_i$, where the real numbers $[\![\mathtt{z}(s,e,k)]\!]$ are understood as constant random variables and each $X_i$ is a random input variable with a user-specified distribution. Currently, PAF supports several well-known distributions out-of-the-box (e.g., uniform, normal, exponential), and the user can also define custom distributions as piecewise functions. For the inductive case $[\![\mathtt{t_1}\ \mathtt{op_m}\ \mathtt{t_2}]\!]$, we put the lessons from Sect. 4 to work. Recall first the probabilistic model from Eq. (3):

$$x \ \mathtt{op_m} \ y = (x \ \text{op} \ y)(1 + \delta), \qquad \delta \sim \mathit{dist}$$

In Sect. 4.1, we showed that *dist* should be taken as the distribution of the actual roundoff errors of the random elements $(x \ \text{op} \ y)$. We therefore define:

$$[\![\mathtt{t_1} \ \mathtt{op_m} \ \mathtt{t_2}]\!] \triangleq ([\![\mathtt{t_1}]\!] \ \text{op} \ [\![\mathtt{t_2}]\!]) \times (1 + \text{err}_{\text{rel}}([\![\mathtt{t_1}]\!] \ \text{op} \ [\![\mathtt{t_2}]\!])) \tag{11}$$

To evaluate the model of Eq. (11), we first use the appropriate closed-form expression Eqs. (5) to (7) derived in Sect. 4 to evaluate the distribution of the random variable $\text{err}_{\text{rel}}([\![\mathtt{t_1}]\!] \ \text{op} \ [\![\mathtt{t_2}]\!])$—or the corresponding p-box as described in Sect. 4.5. We then use Theorem 4 to justify evaluating the multiplication operation in Eq. (11) *independently*—that is to say by using [48]—since the roundoff process is very close to being uncorrelated to the process generating it. The validity of this assumption is also confirmed experimentally by the remarkable agreement of Monte-Carlo simulations with this analytical model.

We now introduce the algorithm for evaluating the model given in Eq. (11). The evaluation performs an in-order (LNR) traversal of the *Abstract Syntax Tree* (AST) of a computation given by our grammar, and it feeds the results to the parent level along the way. At each node, it computes the probabilistic range of the intermediate result using the probabilistic ranges computed for its children nodes (i.e., operands). We first determine whether the operands are independent or not (Ind? branch in the toolflow), and we either apply a cheaper (i.e., no SMT solver invocations) algorithm if they are independent (see below) or a more involved one (see Sect. 6.2) if they are not. We describe our methodology at a generic intermediate computation in the AST of the expression.

We consider two distributions $X$ and $Y$ discretized into DS-structures $DS_X$ and $DS_Y$ (Sect. 3.2), and we want to derive the DS-structure $DS_Z$ for $Z = X \,\text{op}\, Y$, $\text{op} \in \{+, -, \times, \div\}$. Together with the DS-structures of the operands, we also need the traces $trace_X$ and $trace_Y$ containing the history of the operations performed so far, one for each operand. A trace is constructed at each leaf of the AST with the input distributions and their range. It is then propagated to the parent level and populated at each node with the current operation. Such history traces are critical when dealing with dependent operations since they allow us to interrogate an SMT solver about the feasibility of the current operation, as we describe in the next section. When the operands are independent, we simply use the arithmetic operations on independent DS-structures [3].

## 6.2 Computing Probabilistic Ranges for Dependent Operands

When the operands are dependent, we start by assuming that the dependency is unknown. This assumption is sound because the dependency of the operation is included in the set of unknown dependencies, while the result of the operation is no longer a single distribution but a p-box. Due to this "unknown assumption", the CDFs of the output p-box are a very pessimistic over-approximation of the operation, i.e., they are far from each other. Our key insight is to use an

**Algorithm 1.** Dependent Operation $Z = X$ op $Y$

```
1: function DEP_OP($DS_X$, op , $DS_Y$, $trace_X$, $trace_Y$)
2:     $DS_Z = list()$
3:     for all $([x_1, x_2], p_x) \in DS_X$ do
4:         for all $([y_1, y_2], p_y) \in DS_Y$ do
5:             $[z_1, z_2] = [x_1, x_2]$ op $[y_1, y_2]$        ▷ operation between intervals
6:             $[z'_1, z'_2] = SMT.prune([z_1, z_2])$
7:             if $SMT.check(trace_X \wedge trace_Y \wedge [x_1, x_2] \wedge [y_1, y_2])$ is $SAT$ then
8:                 $p_Z$ = unknown-probability
9:             else
10:                $p_Z = 0$
11:            $DS_Z.append(([z'_1, z'_2], p_Z))$
12:    $trace_Z = trace_X \cup trace_Y \cup \{Z = X$ op $Y\}$
13:    return $DS_Z, trace_Z$
```

SMT solver to prune infeasible combinations of intervals from the input DS-structures, which prunes regions of zero probability from the output p-box. This probabilistic pruning using a solver squeezes together the CDFs of the output p-box, often resulting in a much more accurate over-approximation. With the solver, we move from an unknown to a *partially known* dependency between the operands. Currently, PAF supports the Z3 [17] and dReal [23] SMT solvers.

Algorithm 1 shows the pseudocode of our algorithm for computing the probabilistic output range (i.e., DS-structure) for dependent operands. When dealing with dependent operands, interval arithmetic (line 5) might not be as precise as in the independent case. Hence, we use an SMT solver to prune away any over-approximations introduced by interval arithmetic when computing with dependent ranges (line 6); this use of the solver is orthogonal to the one dealing with probabilities. On line 7, we check with an SMT solver whether the current combination of ranges $[x_1, x_2]$ and $[y_1, y_2]$ is compatible with the traces of the operands. If the query is satisfiable, the probability is strictly greater than zero but currently unknown (line 8). If the query is unsatisfiable, we assign a probability of zero to the range in $DS_Z$ (line 10). Finally, we append a new range to the DS-structure $DS_Z$ (line 11). Note that the loops are independent, and hence in our prototype implementation we run them in parallel.

After this algorithm terminates, we still need to assign probability values to all the unknown-probability ranges in $DS_Z$. Since we cannot assign an exact value, we compute a range of potential values $[p_{z_{min}}, p_{z_{max}}]$ instead. This computation is encoded as a *linear programming* routine exactly as in [3].

### 6.3 Computing Conditional Roundoff Error

The final step of our toolflow computes the conditional roundoff error by combining the symbolic affine arithmetic error form of the computation (see Sect. 5) with the probabilistic range analysis described above. The symbolic error form gets maximized conditioned on the results of all the intermediate operations

**Algorithm 2.** Conditional Roundoff Error Computation

```
1: function COND_ERR($DSS, errorForm, confidence$)
2:     $allRanges = list()$
3:     for all $DS_i \in DSS$ do
4:         $focals = sorted(DS_i,\ key = prob,\ order = descending)$
5:         $accumulator = 0$
6:         $ranges = \emptyset$
7:         for all $([x_1, x_2], p_x) \in focals$ do
8:             $accumulator = accumulator + p_x$
9:             $ranges = ranges \cup [x_1, x_2]$
10:            if $accumulator \geq confidence$ then
11:                $allRanges.append(ranges)$
12:                break
13:    $error = maximize(errorForm, allRanges)$
14:    return $error$
```

landing in the given confidence interval (e.g., 99%) of their respective ranges (computed as described in the previous section). Note that conditioning only on the last operation of the computation tree (i.e., the AST root) would lead to extremely pessimistic over-approximation since all the outliers in the intermediate operations would be part of the maximization routine. This would lead to our tool PAF computing pessimistic error bounds typical of worst-case analyzers.

Algorithm 2 shows the pseudocode of the roundoff error computation algorithm. The algorithm takes as input a list *DSS* of DS-structures (one for each intermediate result range in the computation), the generated symbolic error form, and a confidence interval. It iterates over all intermediate DS-structures (line 3), and for each it determines the ranges needed to support the chosen confidence intervals (lines 4–12). In each iteration, it sorts the list of range-probability pairs (i.e., focal elements) of the current DS-structure by their probability value in a descending order (line 4). This is a heuristic that prioritizes the focal elements with most of the probability mass and avoids the unlikely outliers that cause large roundoff errors into the final error computation. With the help of an accumulator (line 8), we keep collecting focal elements (line 9) until the accumulated probability satisfies the confidence interval (line 10). Finally, we maximize the error form conditioned to the collected ranges of intermediate operations (line 13). The maximization is done using the rigorous global optimizer Gelpia [24].

## 7 Experimental Evaluation

We evaluate PAF (version 1.0.0) on the standard FPBench benchmark suite [11, 20] that uses the four basic operations we currently support $\{+, -, \times, \div\}$. Many of these benchmarks were also used in recent related work [36] that we compare against. The benchmarks come from a variety of domains: embedded software (*bsplines*), linear classifications (*classids*), physics computations (*dopplers*), filters (*filters*), controllers (*traincars*, *rigidBody*), polynomial approximations of functions (*sine*, *sqrt*), solving equations (*solvecubic*), and global optimizations (*trids*). Since FPBench has been primarily used for worst-case roundoff error analysis, the benchmarks come with ranges for input variables, but they do not specify input distributions. We instantiate the benchmarks with three well-known distributions for all the inputs: uniform, standard normal distribution, and double exponential (Laplace) distribution with $\sigma = 0.01$ which we will call 'exp'. The normal and exp distributions get truncated to the given range. We assume single-precision floating-point format for all operands and operations.

To assess the accuracy and performance of PAF, we compare it with PrAn (commit 7611679 [10]), the current state-of-the-art tool for automated analysis of probabilistic roundoff errors [36]. PrAn currently supports only uniform and normal distributions. We run all 6 tool configurations and report the best result for each benchmark. We fix the number of intervals in each discretization to 50 to match PrAn. We choose 99% as the confidence interval for the computation of our conditional roundoff error (Sect. 6.3) and of PrAn's probabilistic error. We also compare our probabilistic error bounds against FPTaylor (commit efbbc83 [21]), which performs worst-case roundoff error analysis, and hence it does not take into account the distributions of the input variables. We ran our experiments in parallel on a 4-socket 2.2 GHz 8-core Intel Xeon E5-4620 machine.

Table 2 compares roundoff errors reported by PAF, PrAn, and FPTaylor. PAF outperforms PrAn by computing tighter probabilistic error bounds on almost all benchmarks, occasionally by orders of magnitude. In the case of uniform input distributions, PAF provides tighter bounds for 24 out of 27 benchmarks, for 2 benchmarks the bounds from PrAn are tighter, while for *sqrt* they are the same. In the case of normal input distributions, PAF provides tighter bounds for all the benchmarks. Unlike PrAn, PAF supports probabilistic output range analysis as well. We present these results in the extended version [7].

In Table 2, of particular interest are benchmarks (10 for normal and 18 for exp) where the error bounds generated by PAF for the 99% confidence interval are at least an order of magnitude tighter than the worst-case bounds generated by FPTaylor. For such a benchmark and input distribution, PAF's results inform a user that there is an opportunity to optimize the benchmark (e.g., by reducing precision of floating-point operations) if their use-case can handle at most 1% of inputs generating roundoff errors that exceed a user-provided bound. FPTaylor's results, on the other hand, do not allow for a user to explore such fine-grained trade-offs since they are worst-case and do not take probabilities into account.

In general, we see a gradual reduction of the errors transitioning from uniform to normal to exp. When the input distributions are uniform, there is a significant chance of generating a roundoff error of the same order of magnitude as the worst-case error, since all inputs are equally likely. The standard normal distribution concentrates more than 99% of probability mass in the interval $[-3, 3]$, resulting in the *long tail* phenomenon, where less than 0.5% of mass spreads in the interval $[3, \infty]$. When the normal distribution gets truncated in a neighborhood of zero (e.g., $[0, 1]$ for *bsplines* and *filters*) nothing changes with respect to the uniform case—there is still a high chance of committing errors close to the worst-case.

**Table 2.** Roundoff error bounds reported by PAF, PrAn, and FPTaylor given uniform (uni), normal (norm), and Laplace (exp) input distributions. We set the confidence interval to 99% for PAF and PrAn, and mark the smallest reported roundoff errors for each benchmark in bold. Asterisk (*) highlights a difference of more than one order of magnitude between PAF and FPTaylor.

| Benchmark | Uniform | | Normal | | Exp | FpTaylor |
|---|---|---|---|---|---|---|
| | PAF | PrAn | PAF | PrAn | PAF | |
| bspline0 | **5.71e−08** | 6.12e−08 | **5.71e−08** | 6.12e−08 | **5.71e−08** | 5.72e−08 |
| bspline1 | **1.86e−07** | 2.08e−07 | **1.86e−07** | 2.08e−07 | **6.95e−08** | 1.93e−07 |
| bspline2 | **1.94e−07** | 2.13e−07 | **1.94e−07** | 2.13e−07 | **2.11e−08** | 2.10e−07 |
| bspline3 | **4.22e−08** | 4.65e−08 | **4.22e−08** | 4.65e−08 | **7.62e−12***| 4.22e−08 |
| classids0 | **6.93e−06** | 8.65e−06 | **4.45e−06** | 8.64e−06 | **1.70e−06** | 6.85e−06 |
| classids1 | **3.71e−06** | 4.63e−06 | **2.68e−06** | 4.62e−06 | **7.62e−07** | 3.62e−06 |
| classids2 | **5.23e−06** | 7.32e−06 | **3.85e−06** | 7.32e−06 | **1.46e−06** | 5.15e−06 |
| doppler1 | **7.95e−05** | 1.17e−04 | **5.08e−07***| 1.17e−04 | **4.87e−07***| 6.10e−05 |
| doppler2 | **1.43e−04** | 2.45e−04 | **6.61e−07***| 2.45e−04 | **6.28e−07***| 1.11e−04 |
| doppler3 | **4.55e−05** | 5.12e−05 | **9.11e−07***| 5.12e−05 | **8.95e−07***| 3.41e−05 |
| filter1 | **1.25e−07** | 2.03e−07 | **1.25e−07** | 2.03e−07 | **5.43e−09***| 1.25e−07 |
| filter2 | **7.93e−07** | 1.01e−06 | **6.13e−07** | 1.01e−06 | **2.90e−08***| 7.93e−07 |
| filter3 | **2.34e−06** | 2.86e−06 | **2.05e−06** | 2.87e−06 | **1.09e−07***| 2.23e−06 |
| filter4 | **4.15e−06** | 5.20e−06 | **4.15e−06** | 5.20e−06 | **4.61e−07** | 3.81e−06 |
| rigidbody1 | 1.74e−04 | **1.58e−04** | **6.14e-06***| 1.58e−04 | **4.80e−07***| 1.58e−04 |
| rigidbody2 | 1.96e−02 | **9.70e−03** | **5.99e-05***| 9.70e−03 | **9.55e−07***| 1.94e−02 |
| sine | **2.37e−07** | 2.40e−07 | **2.37e−07** | 2.40e−07 | **1.49e−08***| 2.38e−07 |
| solvecubic | **1.78e−05** | 1.83e−05 | **6.84e−06** | 1.83e−05 | **2.76e−06** | 1.60e−05 |
| sqrt | **1.54e−04** | **1.54e-04** | **1.10e−06***| 1.54e−04 | **2.46e−07***| 1.51e−04 |
| traincars1 | **1.76e−03** | 1.96e−03 | **8.26e−04** | 1.96e−03 | **4.50e−04** | 1.74e−03 |
| traincars2 | **1.04e−03** | 1.36e−03 | **3.61e−04** | 1.36e−03 | **2.83e−05***| 9.46e−04 |
| traincars3 | **1.75e−02** | 2.29e−02 | **9.56e−03** | 2.29e−02 | **8.95e−04***| 1.80e−02 |
| traincars4 | **1.81e−01** | 2.30e−01 | **8.87e−02** | 2.30e−01 | **7.33e−03***| 1.81e−01 |
| trid1 | **6.01e−03** | 6.03e−03 | **1.58e−05***| 6.03e−03 | **1.58e−05***| 6.06e−03 |
| trid2 | **1.03e−02** | 1.17e−02 | **2.42e−05***| 1.17e−02 | **2.43e−05***| 1.03e−02 |
| trid3 | **1.75e−02** | 1.95e−02 | **6.80e−05***| 1.95e−02 | **6.77e−05***| 1.75e−02 |
| trid4 | **2.69e−02** | 2.88e−02 | **2.64e−04***| 3.03e−02 | **2.64e−04***| 2.66e−02 |

However, when the normal distribution gets truncated to a wider range (e.g., $[-100, 100]$ for *trids*), then the outliers causing large errors are very rare events, not included in the 99% confidence interval. The exponential distribution further compresses the 99% probability mass in the tiny interval $[-0.01, 0.01]$, so the long tails effect is common among all the benchmarks.

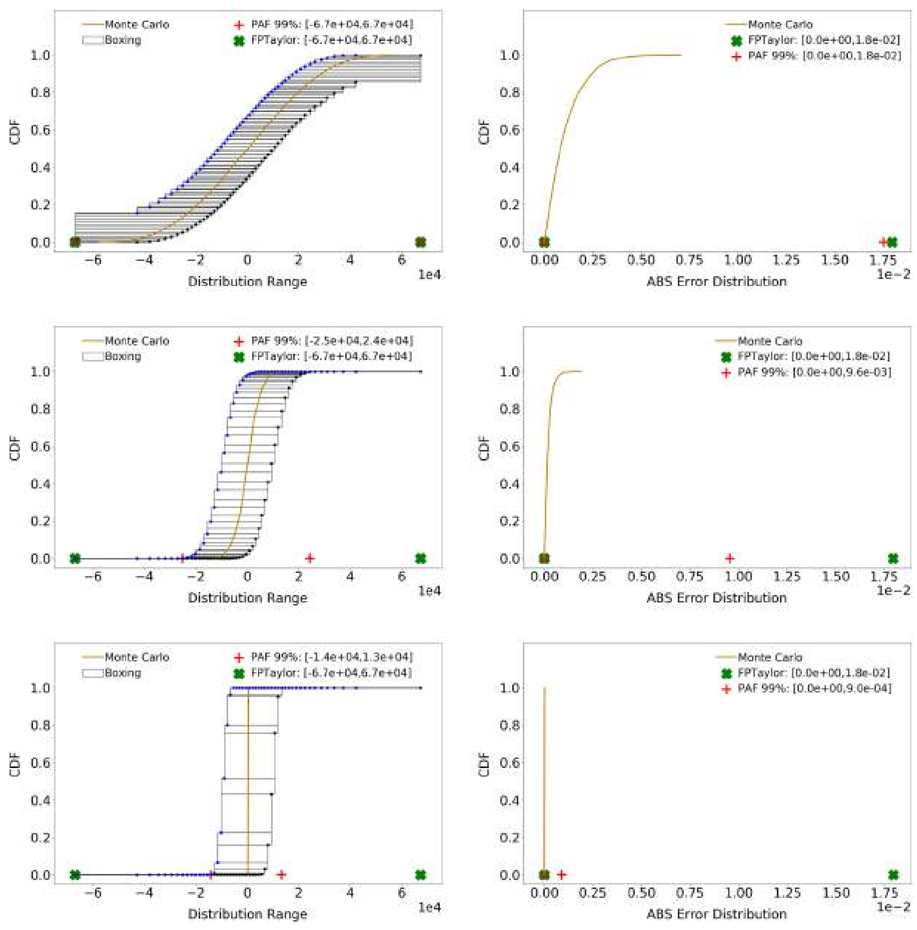


**Fig. 4.** CDFs of the range (left) and error (right) distributions for the benchmark *traincars3* for uniform (top), normal (center), and exp (bottom).

The runtimes of PAF vary between 10 min for small benchmarks, such as *bsplines*, to several hours for benchmarks with more than 30 operations, such as *trid4*; they are always less than two hours, except for *trids* with 11 h and *filters* with 6 h. The runtime of PAF is usually dominated by Z3 invocations, and the long runtimes are caused by numerous Z3 timeouts that the respective benchmarks induce. The runtimes of PrAn are comparable to PAF since they are always less than two hours, except for *trids* with 3 h, *sqrt* with 3 h, and *sine* with 11 h. Note that neither PAF nor PrAn are memory intensive.

To assess the quality of our rigorous (i.e., sound) results, we implement Monte Carlo sampling to generate both roundoff error and output range distributions. The procedure consists of randomly sampling from the provided input distributions, evaluating the floating-point computation in both the specified and high-precision (e.g., double-precision) floating-point regimes to measure the roundoff error, and finally partitioning the computed errors into bins to get an approximation (i.e., histogram) of the PDF. Of course, Monte Carlo sampling does not provide rigorous bounds, but is a useful tool to assess how far the rigorous bounds computed statically by PAF are from an empirical measure of the error.

Figure 4 shows the effects of the input distributions on the output and roundoff error ranges of the *traincars3* benchmark. In the error graphs (right column), we show the Monte Carlo sampling evaluation (yellow line) together with the error bounds from PAF with 99% confidence interval (red plus symbol) and FPTaylor's worst-case bounds (green crossmark). In the range graphs (left column), we also plot PAF's p-box over-approximations. We can observe that in the case of uniform inputs the computed p-boxes overlap at the extrema of the output range. This phenomenon makes it impossible to distinguish between 99% and 100% confidence intervals, and hence as expected the bound reported by PAF is almost identical to FPTaylor's. This is not the case for normal and exponential distributions, where PAF can significantly improve both the output range and error bounds over FPTaylor. This again illustrates how pessimistic the bounds from worst-case tools can be when the information about the input distributions is not taken into account. Finally, the graphs illustrate how the p-boxes and error bounds from PAF follow their respective empirical estimations.

## 8 Related Work

Our work draws inspiration from *probabilistic affine arithmetic* [3,4], which aims to bound probabilistic uncertainty propagated through a computation; a similar goal to our probabilistic range analysis. This was recently extended to polynomial dependencies [45]. On the other hand, PAF detects any non-linear dependency supported by the SMT solver. While these approaches show how to bound moments, we do not consider moments but instead compute conditional roundoff error bounds, a concern specific to the analysis of floating-point computations. Finally, the concentration of measure inequalities [4,45] provides bounds for (possibly very large) problems that can be expressed as sums of random variables, for example multiple increments of a noisy dynamical system, but are unsuitable for typical floating-point computations (such as FPBench benchmarks).

The most similar approach to our work is the recent static probabilistic roundoff error analysis called PrAn [36]. PrAn also builds on [3], and inherits the same limitations in dealing with dependent operations. Like us, PrAn hinges on a discretization scheme that builds p-boxes for both the input and error distributions and propagates them through the computation. The question of how these p-boxes are chosen is left open in the PrAn approach. In contrast, we take the input variables to be user-specified random variables, and show how the distribution of each error term can be computed directly and exactly from the random variables generating it (Sect. 4). Furthermore, unlike PrAn, PAF leverages the non-correlation between random variables and the corresponding error distribution (Sect. 4.4). Thus, PAF performs the rounding in Eq. (3) as an *independent* operation. Putting these together leads to PAF computing tighter probabilistic roundoff error bounds than PrAn, as our experiments show (Sect. 7).

The idea of using a probabilistic model of rounding errors to analyze *deterministic* computations can be traced back to Von Neumann and Goldstine [51]. Parker's so-called 'Monte Carlo arithmetic' [41] is probably the most detailed description of this approach. We, however, consider *probabilistic* computations. For this reason, the famous critique of the probabilistic approach to roundoff errors [29] does not apply to this work. Our preliminary report [9] presents some early ideas behind this work, including Eqs. (5) and (7) and a very rudimentary range analysis. However, this early work manipulated distributions *unsoundly*, could not handle any repeated variables, and did not provide any roundoff error analysis. Recently, probabilistic roundoff error models have also been investigated using the concentration of measure inequalities [27,28]. Interestingly, this means that the distribution of errors in Eq. (3) can be left almost completely unspecified. However, as in the case of related work from the beginning of this section [4,45], concentration inequalities are very ill-suited to the applications captured by the FPBench benchmark suite.

Worst-case analysis of roundoff errors has been an active research area with numerous published approaches [12–16,18,22,33,35,37,38,46,47,50]. Our symbolic affine arithmetic used in PAF (Sect. 5) evolved from rigorous affine arithmetic [14] by keeping the coefficients of the noise terms symbolic, which often leads to improved precision. These symbolic terms are very similar to the first-order Taylor approximations of the roundoff error expressions used in FPTaylor [46,47]. Hence, PAF with the 100% confidence interval leads to the same worst-case roundoff error bounds as computed by FPTaylor (Sect. 7).

**Acknowledgments.** We thank Ian Briggs and Mark Baranowski for their generous and prompt support with Gelpia. We also thank Alexey Solovyev for his detailed feedback and suggestions for improvements. Finally, we thank the anonymous reviewers for their insightful reviews that improved our final version, and program chairs for carefully guiding the review process.

## References

1. Bornholt, J.: Abstractions and techniques for programming with uncertain data. Undergraduate honours thesis, Australian National University (2013)
2. Bornholt, J., Mytkowicz, T., McKinley, K.S.: Uncertain <T>: a first-order type for uncertain data. In: ASPLOS (2014)
3. Bouissou, O., Goubault, E., Goubault-Larrecq, J., Putot, S.: A generalization of p-boxes to affine arithmetic. Computing **89**, 189–201 (2012). https://doi.org/10.1007/s00607-011-0182-8
4. Bouissou, O., Goubault, E., Putot, S., Chakarov, A., Sankaranarayanan, S.: Uncertainty propagation using probabilistic affine forms and concentration of measure inequalities. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 225–243. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_13

5. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: POPL (2017)
6. Comba, J.L.D., Stolfi, J.: Affine arithmetic and its applications to computer graphics. In: SIBGRAPI (1993)
7. Constantinides, G., Dahlqvist, F., Rakamarić, Z., Salvia, R.: Rigorous roundoff error analysis of probabilistic floating-point computations (2021). arXiv:2105.13217
8. Dahlqvist, F., Kozen, D.: Semantics of higher-order probabilistic programs with conditioning. In: POPL (2019)
9. Dahlqvist, F., Salvia, R., Constantinides, G.A.: A probabilistic approach to floating-point arithmetic. In: ASILOMAR (2019). Non-peer-reviewed extended abstract
10. Daisy. https://github.com/malyzajko/daisy
11. Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., Tatlock, Z.: Toward a standard benchmark format and suite for floating-point analysis. In: Bogomolov, S., Martel, M., Prabhakar, P. (eds.) NSV 2016. LNCS, vol. 10152, pp. 63–77. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54292-8_6
12. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 270–287. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_15
13. Darulova, E., Kuncak, V.: Trustworthy numerical computation in Scala. In: OOPSLA (2011)
14. Darulova, E., Kuncak, V.: Sound compilation of reals. In: POPL (2014)
15. Das, A., Briggs, I., Gopalakrishnan, G., Krishnamoorthy, S., Panchekha, P.: Scalable yet rigorous floating-point error analysis. In: SC (2020)
16. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. **37**, 1–20 (2010)
17. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 53–69. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04570-7_6
19. Ferson, S., Kreinovich, V., Grinzburg, L., Myers, D., Sentz, K.: Constructing probability boxes and dempster-shafer structures. Technical report, Sandia National Lab (2015)
20. FPBench: standards and benchmarks for floating-point research. https://fpbench.org
21. FPTaylor. https://github.com/soarlab/fptaylor
22. Fu, Z., Bai, Z., Su, Z.: Automated backward error analysis for numerical code. In: OOPSLA (2015)
23. Gao, S., Kong, S., Clarke, E.M.: dReal: an SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 208–214. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_14
24. Gelpia: a global optimizer for real functions. https://github.com/soarlab/gelpia
25. Glasserman, P.: Monte Carlo Methods in Financial Engineering. Springer, Heidelberg (2013). https://doi.org/10.1007/978-0-387-21617-1
26. Higham, N.J.: Accuracy and Stability of Numerical Algorithms. SIAM (2002)
27. Higham, N.J., Mary, T.: A New Approach to Probabilistic Rounding Error Analysis. SISC (2019)

28. Ipsen, I.C.F., Zhou, H.: Probabilistic error analysis for inner products. SIMAX (2019)
29. Kahan, W.: The improbability of probabilistic error analyses for numerical computations (1996)
30. Kajiya, J.T.: The rendering equation. In: SIGGRAPH (1986)
31. Kozen, D.: Semantics of probabilistic programs. In: JCSS (1981)
32. Landau, D.P., Binder, K.: A Guide to Monte Carlo Simulations in Statistical Physics. Cambridge University Press, Cambridge (2014)
33. Lee, W., Sharma, R., Aiken, A.: Verifying bit-manipulations of floating-point. In: PLDI (2016)
34. Lepage, G.P.: VEGAS – an adaptive multi-dimensional integration program. Technical report, Cornell (1980)
35. Linderman, M.D., Ho, M., Dill, D.L., Meng, T.H., Nolan, G.P.: Towards program optimization through automated analysis of numerical precision. In: CGO (2010)
36. Lohar, D., Prokop, M., Darulova, E.: Sound probabilistic numerical error analysis. In: IFM (2019)
37. Magron, V., Constantinides, G., Donaldson, A.: Certified roundoff error bounds using semidefinite programming. TOMS **43**, 1–31 (2017)
38. Martel, M.: RangeLab: a static-analyzer to bound the accuracy of finite-precision computations. In: SYNASC (2011)
39. Microprocessor Standards Committee of the IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic (2019)
40. Moore, R.E.: Interval Analysis. Prentice-Hall, Hoboken (1966)
41. Parker, D.S., Pierce, B., Eggert, P.R.: Monte Carlo arithmetic: how to gamble with floating point and win. Comput. Sci. Eng. **2**, 58–68 (2000)
42. Press, W.H., Farrar, G.R.: Recursive stratified sampling for multidimensional Monte Carlo integration. Comput. Phys. **4**, 190–195 (1990)
43. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C. Cambridge University Press, Cambridge (1988)
44. Rothschild, M., Stiglitz, J.E.: Increasing risk: I. A definition. J. Econ. Theory **2**, 225–243 (1970)
45. Sankaranarayanan, S., Chou, Y., Goubault, E., Putot, S.: Reasoning about uncertainties in discrete-time dynamical systems using polynomial forms. In: NeurIPS (2020)
46. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. TOPLAS **41**, 1–39 (2018)
47. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. In: FM (2015)
48. Malik, H.J.: The Algebra of Random Variables (Springer, MD). Wiley (1979)
49. Stolfi, J., Figueiredo, L.H.D.: Self-validated numerical methods and applications. In: IMPA (1997)
50. Titolo, L., Feliú, M.A., Moscato, M., Muñoz, C.A.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: VMCAI (2018)
51. Von Neumann, J., Goldstine, H.H.: Numerical inverting of matrices of high order. Bull. Am. Math. Soc. **53**, 1021–1099 (1947)

# Model-Free Reinforcement Learning for Branching Markov Decision Processes

Ernst Moritz Hahn[1], Mateo Perez[2], Sven Schewe[3], Fabio Somenzi[2], Ashutosh Trivedi[2](✉), and Dominik Wojtczak[3]

[1] University of Twente, Enschede, The Netherlands
[2] University of Colorado Boulder, Boulder, USA
ashutosh.trivedi@colorado.edu
[3] University of Liverpool, Liverpool, UK

**Abstract.** We study reinforcement learning for the optimal control of Branching Markov Decision Processes (BMDPs), a natural extension of (multitype) Branching Markov Chains (BMCs). The state of a (discrete-time) BMCs is a collection of entities of various types that, while spawning other entities, generate a payoff. In comparison with BMCs, where the evolution of a each entity of the same type follows the same probabilistic pattern, BMDPs allow an external controller to pick from a range of options. This permits us to study the best/worst behaviour of the system. We generalise model-free reinforcement learning techniques to compute an optimal control strategy of an unknown BMDP in the limit. We present results of an implementation that demonstrate the practicality of the approach.

## 1 Introduction

Branching Markov Chains (BMCs), also known as Branching Processes, are natural models of population dynamics and parallel processes. The state of a BMC consists of entities of various types, and many entities of the same type may coexist. Each entity can branch in a single step into a (possibly empty) set of entities of various types while disappearing itself. This assumption is natural, for instance, for annual plants that reproduce only at a specific time of the year, or for bacteria, which either split or die. An entity may spawn a copy of itself, thereby simulating the continuation of its existence.

The offspring of an entity is chosen at random among options according to a distribution that depends on the type of the entity. The type captures significant differences between entities. For example, stem cells are very different from

This work was supported by the Engineering and Physical Sciences Research Council through grant EP/P020909/1 and by the National Science Foundation through grant 2009022. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101032464 (SyGaST) and 864075 (CAESAR).

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 651–673, 2021.
https://doi.org/10.1007/978-3-030-81688-9_30

regular cells; parallel processes may be interruptible or have different privileges. The type may reflect characteristics of the entities such as their age or size.

Although entities coexist, the BMC model assumes that there is no interaction between them. Thus, how an entity reproduces and for how long it lives is the same as if it were the only entity in the system. This assumption greatly improves the computational complexity of the analysis of such models and is appropriate when the population exists in an environment that has virtually unlimited resources to sustain its growth. This is a common situation that holds when a species has just been introduced into an environment, in an early stage of an epidemic outbreak, or when running jobs in cloud computing.

BMCs have a wide range of applications in modelling various physical phenomena, such as nuclear chain reactions, red blood cell formation, population genetics, population migration, epidemic outbreaks, and molecular biology. Many examples of BMC models used in biological systems are discussed in [12].

Branching Markov Decision Processes (BMDPs) extend BMCs by allowing a controller to choose the branching dynamics for each entity. This choice is modelled as nondeterministic, instead of random. This extension is analogous to how Markov Decision Processes (MDPs) generalise Markov chains (MCs) [24]. Allowing an external controller to select a mode of branching allows us to study the best/worst behaviour of the examined model.

As a motivating example, let us discuss a simple model of cloud computing. A computation may be divided into tasks in order to finish it faster, as each server may have different computational power. Since the computation of each task depends on the previous one, the total running time is the sum of the running times of each spawned task as well as the time needed to split and merge the result of each computation into the final solution. As we shall see, the execution of each task is not guaranteed to be successful and is subject to random delays. Specifically, let us consider the following model with two different types ($T$ and $S$), and two actions ($a_1$ and $a_2$). This BMDP consists of the main task, $T$, that may be split (action $a_1$) into three smaller tasks, for simplicity assumed to be of the same type $S$, and this split and merger of the intermediate results takes 1 hour (1h). Alternatively (action $a_2$), we can execute the whole task $T$ on the main server, but it will be slow (8 h). Task $S$ can (action $a_1$) be run on a reliable server in 1.6 h or (action $a_2$) an unreliable one that finishes after 1 h (irrespective of whether or not the computation is completed successfully), but with a 40% chance we need to rerun this task due to the server crashing. We can represent this model formally as:

$$\begin{aligned} &T \xrightarrow{a_1} SSS \qquad [1\text{h}] \qquad && S \xrightarrow{a_1} \epsilon && [1.6\text{h}] \\ &T \xrightarrow{a_2} \epsilon \qquad [8\text{h}] \qquad && S \xrightarrow{a_2} 40\% : S \text{ or } 60\% : \epsilon && [1\text{h}] \end{aligned}$$

We would like to know the infimum of the expected running time (i.e. the expected running time when optimal decisions are made) of task $T$. In this case the optimal control is to pick action $a_1$ first and then actions $a_1$ for all tasks S with a total running time of 5.8 h. The expected running time when picking actions $a_2$ for $S$ instead would be $1 + 3 \cdot 1/0.6 = 6$ [hours].

Let us now assume that the execution of tasks $S$ for action $a_1$ may be interrupted with probability 30% by a task of higher priority (type $H$). Moreover, these $H$ tasks may be further interrupted by tasks with even higher priority (to simplify matters, again modelled by type $H$). The computation time of $T$ is prolonged by 0.1 h for each $H$ spawned. Our model then becomes:

$$\begin{array}{llllllll}
T \xrightarrow{a_1} SSS & [1\text{h}] & S \xrightarrow{a_1} 30\% : H \text{ or } 70\% : \epsilon & [1.6\text{h}] & H \xrightarrow{*} & 30\% : HH \text{ or} & \\
T \xrightarrow{a_2} \epsilon & [8\text{h}] & S \xrightarrow{a_2} 40\% : S \text{ or } 60\% : \epsilon & [1\text{h}] & & 70\% : \epsilon & [0.1\text{h}]
\end{array}$$

As we shall see, the expected total running time of $H$ can be calculated by solving the equation $x = 0.3(x + x) + 0.1$, which gives $x = 0.25$ [hour]. So the expected running time of $S$ using action $a_1$ increases by $0.3 \cdot 0.25 = 0.075$ [hour]. This is enough for the optimal strategy of running $S$ to become $a_2$. Note that if the probability of $H$ being interrupted is at least 50% then the expected running time of $H$ becomes $\infty$.

When dealing with a real-life process, it is hard to come up with a (probabilistic and controlled) model that approximates it well. This requires experts to analyse all possible scenarios and estimate the probability of outcomes in response to actions based on either complex calculations or the statistical analysis of sufficient observational data. For instance, it is hard to estimate the probability of an interrupt $H$ occurring in the model above without knowing which server will run the task, its usual workload and statistics regarding the priorities of the tasks it executes. Even if we do this estimation well, unexpected or rare events may happen that would require us to recalibrate the model as we observe the system under our control.

Instead of building such a model explicitly first and fixing the probabilities of possible transitions in the system based on our knowledge of the system or its statistics, we advocate the use of reinforcement learning (RL) techniques [27] that were successfully applied to finding optimal control for finite-state Markov Decision Processes (MDPs). Q-learning [30] is a well-studied model-free RL approach to compute an optimal control strategy without knowing about the model apart from its initial state and the set of actions available in each of its states. It also has the advantage that the learning process converges to the optimal control while exploiting along the way what it already knows. While the formulation of the Q-learning algorithm for BMDPs is straightforward, the proof that it works is not. This is because, unlike the MDPs with discounted rewards for which the original Q-learning algorithm was defined, our model does not have an explicit contraction in each step, nor does boundedness of the optimal values or one-step updates hold. Similarly, one cannot generalise the result from [11] that estimates the time needed for the Q-learning algorithm to converge within $\epsilon$ of the optimal values with high probability for finite-state MDPs.

## 1.1 Related Work

The simplest model of BMCs are Galton-Watson processes [31], discrete-time models where all entities are of the same type. They date as far back as 1845 [14] and were used to explain why some aristocratic family surnames became extinct. The generalisation of this model to multiple types of entities was first studied in 1940s by Kolmogorov and Sevast'yanov [17]. For an overview of the results known for BMCs, see e.g. [13] and [12]. The precise computational complexity of decision problems about the probabilities of extinction of an arbitrary BMC was first established in [9]. The problem of checking if a given BMC terminates almost surely was shown in [5] to be strongly polynomial. The probability of acceptance of a run of a BMC by a deterministic parity tree automaton was studied in [4] and shown to be computable in PSPACE and in polynomial time for probabilities 0 or 1. In [16] a generalisation of the BMCs was considered that allowed for limited synchronisation of different tasks.

BMDPs, a natural generalisation of BMCs to a controlled setting, have been studied in the OR literature e.g., [23,26]. Hierarchical MDPs (HMDPs) [10] are a special case of BMDPs where there are no cycles in the offspring graph (equivalently, no cyclic dependency between types). BMDPs and HMDPs have found applications in manpower planning [29], controlled queuing networks [2, 15], management of livestock [20], and epidemic control [1,25], among others. The focus of these works was on optimising the expected average, or the discounted reward over a run of the process, or optimising the population growth rate. In [10] the decision problem whether the optimal probability of termination exceeds a threshold was studied: it was shown to be solvable in PSPACE and at least as hard as the square-root sum problem, but one can determine if the optimal probability is 0 or 1 in polynomial time. In [7], it was shown that the approximation of the optimal probability of extinction for BMDPs can be done in polynomial time. The computational complexity of computing the optimal expected total cost before extinction for BMDPs follows from [8] and was shown there to be computable in polynomial time via a linear program formulation. The problem of maximising the probability of reaching a state with an entity of a given type for BMDPs was studied in [6]. In [28] an extension of BMDPs with real-valued clocks and timing constraints on productions was studied.

### 1.2 Summary of the Results

We show that an adaptation of the Q-learning algorithm converges almost surely to the optimal values for BMDPs under mild conditions: all costs are positive and each Q-value is selected for update independently at random. We have implemented the proposed algorithm in the tool Mungojerrie [21] and tested its performance on small examples to demonstrate its efficiency in practice. To the best of our knowledge, this is the first time model-free RL has been used for the analysis of BMDPs.

## 2 Problem Definitions

### 2.1 Preliminaries

We denote by $\mathbb{N}$ the set of non-negative integers, by $\mathbb{R}$ the set of reals, by $\mathbb{R}_+$ the set of positive reals, and by $\mathbb{R}_{\geq 0}$ the set of non-negative reals. We let

$\widetilde{\mathbb{R}}_+ = \mathbb{R}_+ \cup \{\infty\}$, and $\widetilde{\mathbb{R}}_{\geq 0} = \mathbb{R}_{\geq 0} \cup \{\infty\}$. We denote by $|X|$ the cardinality of a set $X$ and by $X^*$ ($X^\omega$) the set of all possible finite (infinite) sequences of elements of $X$. Finite sequences are also called lists.

*Vectors and Lists.* We use $\bar{x}, \bar{y}, \bar{c}$ to denote vectors and $\bar{x}_i$ or $\bar{x}(i)$ to denote its $i$-th entry. We let $\bar{0}$ denote a vector with all entries equal to 0; its size may vary depending on the context. Likewise $\bar{1}$ is a vector with all entries equal to 1. For vectors $\bar{x}, \bar{y} \in \widetilde{\mathbb{R}}^n_{\geq 0}$, $\bar{x} \leq \bar{y}$ means $x_i \leq y_i$ for every $i$, and $\bar{x} < \bar{y}$ means $\bar{x} \leq \bar{y}$ and $x_i \neq y_i$ for some $i$. We also make use of the infinity norm $\|\bar{x}\|_\infty = \max_i |\bar{x}(i)|$.

We use $\alpha, \beta, \gamma$ to denote finite lists of elements. For a list $\alpha = a_1, a_2, \ldots, a_k$ we write $\alpha_i$ for the $i$-th element $a_i$ of list $\alpha$ and $|\alpha|$ for its length. For two lists $\alpha$ and $\beta$ we write $\alpha \cdot \beta$ for their concatenation. The empty list is denoted by $\epsilon$.

*Probability Distributions.* A *finite discrete probability distribution* over a countable set $Q$ is a function $\mu : Q \to [0,1]$ such that $\sum_{q \in Q} \mu(q) = 1$ and its support set $supp(\mu) = \{q \in Q \mid \mu(q) > 0\}$ is finite. We say that $\mu \in \mathcal{D}(Q)$ is a *point distribution* if $\mu(q) = 1$ for some $q \in Q$.

*Markov Decision Processes.* Markov decision processes [24], are a well-studied formalism for systems exhibiting nondeterministic and probabilistic behaviour.

**Definition 1.** *A* Markov decision process *(MDP) is a tuple* $\mathcal{M} = (S, A, p, c)$ *where:*

- $S$ *is the set of* states*;*
- $A$ *is the set of* actions*;*
- $p : S \times A \to \mathcal{D}(S)$ *is a partial function called the* probabilistic transition function*; and*
- $c : S \times A \to \mathbb{R}$ *is the* cost function*.*

We say that an MDP $\mathcal{M}$ is *finite* (*discrete*) if both $S$ and $A$ are finite (countable). We write $A(s)$ for the set of actions available at $s$, i.e., the set of actions $a$ for which $p(s,a)$ is defined. In an MDP $\mathcal{M}$, if the current state is $s$, then one of the actions in $A(s)$ is chosen nondeterministically. If the chosen action is $a$ then the probability of reaching state $s' \in S$ in the next step is $p(s,a)(s')$ and the cost incurred is $c(s,a)$.

## 2.2 Branching Markov Decision Processes

We are now ready to define (multitype) BMDPs.

**Definition 2.** *A* branching Markov decision process *(BMDP) is a tuple* $\mathcal{B} = (P, A, p, c)$ *where:*

- $P$ *is a finite set of* types*;*
- $A$ *is a finite set of* actions*;*
- $p : P \times A \to \mathcal{D}(P^*)$ *is a partial function called the* probabilistic transition function *where every* $\mathcal{D}(\cdot)$ *is a finite discrete probability distribution; and*

– $c : P \times A \to \mathbb{R}_+$ *is the* cost function.

We write $A(q)$ for the set of actions available to an entity of type $q \in P$, i.e., the set of actions $a$ for which $p(q, a)$ is defined. A *Branching Markov Chain (BMC)* is simply a BMDP with just one action available for each type.

Let us first describe informally how BMDPs evolve. A state of a BMDP $\mathcal{B}$ is a list of elements of $P$ that we call *entities*. A BMDP starts at some initial configuration, $\alpha^0 \in P^*$, and the controller picks for one of the entities one of the actions available to an entity of its type. In the new configuration $\alpha^1$, this one entity is replaced by the list of new entities that it spawned. This list is picked according to the probability distribution $p(q, a)$ that depends both on the type of the entity, $q$, and the action, $a$, performed on it by the controller. The process proceeds in the same manner from $\alpha^1$, moving to $\alpha^2$, and from there to $\alpha^3$, etc. Once the state $\epsilon$ is reached, i.e., when no entities are present in the system, the process stays in that state forever.

**Definition 3 (Semantics of BMDP).** *The semantics of a BMDP* $\mathcal{B} = (P, A, p, c)$ *is an MDP* $\mathcal{M}_\mathcal{B} = (\mathit{States}_\mathcal{B}, \mathit{Actions}_\mathcal{B}, \mathit{Prob}_\mathcal{B}, \mathit{Cost}_\mathcal{B})$ *where:*

– $\mathit{States}_\mathcal{B} = P^*$ *is the set of states;*
– $\mathit{Actions}_\mathcal{B} = \mathbb{N} \times A$ *is the set of actions;*
– $\mathit{Prob}_\mathcal{B} : \mathit{States}_\mathcal{B} \times \mathit{Actions}_\mathcal{B} \to \mathcal{D}(\mathit{States}_\mathcal{B})$ *is the probabilistic transition function such that, for* $\alpha \in \mathit{States}_\mathcal{B}$ *and* $(i, a) \in \mathit{Actions}_\mathcal{B}$*, we have that* $\mathit{Prob}_\mathcal{B}(\alpha, (i, a))$ *is defined when* $i \leq |\alpha|$ *and* $a \in A(\alpha_i)$*; moreover*
$$\mathit{Prob}_\mathcal{B}(\alpha, (i, a))(\alpha_1 \ldots \alpha_{i-1} \cdot \beta \cdot \alpha_{i+1} \ldots) = p(\alpha_i, a)(\beta),$$
*for every* $\beta \in P^*$ *and* $0$ *in all other cases.*
– $\mathit{Cost}_\mathcal{B} : \mathit{States}_\mathcal{B} \times \mathit{Actions}_\mathcal{B} \to \mathbb{R}_+$ *is the cost function such that*
$$\mathit{Cost}_\mathcal{B}(\alpha, (i, a)) = c(\alpha_i, a).$$

For a given BMDP $\mathcal{B}$ and states $\alpha \in \mathit{States}_\mathcal{B}$, we denote by $\mathit{Actions}_\mathcal{B}(\alpha)$ the set of actions $(i, a) \in \mathit{Actions}_\mathcal{B}$, for which $\mathit{Prob}_\mathcal{B}(\alpha, (i, a))$ is defined.

Note that our semantics of BMDPs assumes an explicit listing of all the entities in a particular order similar to [10]. One could, instead, define this as a multi-set or simply a vector just counting the number of occurrences of each entity as in [23]. As argued in [10], all these models are equivalent to each other. Furthermore, we assume that the controller expands a single entity of his choice at the time rather all of them being expanded simultaneously. As argued in [32], that makes no difference for the optimal values of the expected total cost that we study in this paper, provided that all transitions' costs are positive.

## 2.3 Strategies

A *path* of a BMDP $\mathcal{B}$ is a finite or infinite sequence
$$\begin{aligned}\pi = \alpha^0, ((i_1, a_1), \alpha^1), ((i_2, a_2), \alpha^2), ((i_3, a_3), \alpha^3), \ldots \\ \in \mathit{States}_\mathcal{B} \times ((\mathit{Actions}_\mathcal{B} \times \mathit{States}_\mathcal{B})^* \cup (\mathit{Actions}_\mathcal{B} \times \mathit{States}_\mathcal{B})^\omega),\end{aligned}$$

consisting of the initial state and a finite or infinite sequence of action and state pairs, such that $\mathit{Prob}_{\mathcal{B}}(\alpha^j, (i_j, a_j))(\alpha^{j+1}) > 0$ for any $0 \leq j \leq |\pi|$, where $|\pi|$ is the number of actions taken during path $\pi$. ($|\pi| = \infty$ if the path is infinite.) For a path $\pi$, we denote by $\pi_{A(j)} = (i_j, a_j)$ the $j$-th action taken along path $\pi$, by $\pi_{S(j)}(= \alpha^j)$ the $j$-th state visited, where $\pi_{S(0)}(= \alpha^0)$ is the initial state, and by $\pi(j)(= \alpha^0, ((i_1, a_1), \alpha^1), \ldots, ((i_j, a_j), \alpha^j))$ the first $j$ action-state pairs of $\pi$.

We call a path of infinite (finite) length a *run* (*finite path*). We write $\mathit{Runs}_{\mathcal{B}}$ ($\mathit{FPath}_{\mathcal{B}}$) for the sets of all runs (finite paths) and $\mathit{Runs}_{\mathcal{B},\alpha}$ ($\mathit{FPath}_{\mathcal{B},\alpha}$) for the sets of all runs (finite paths) that start at a given initial state $\alpha \in \mathit{States}_{\mathcal{B}}$, i.e., paths $\pi$ with $\pi_{S(0)} = \alpha$. We write $\mathit{last}(\pi)$ for the last state of a finite path $\pi$.

A *strategy* in BMDP $\mathcal{B}$ is a function $\sigma : \mathit{FPath}_{\mathcal{B}} \to \mathcal{D}(\mathit{Actions}_{\mathcal{B}})$ such that, for all $\pi \in \mathit{FPath}_{\mathcal{B}}$, $\mathit{supp}(\sigma(\pi)) \subseteq \mathit{Actions}_{\mathcal{B}}(\mathit{last}(\pi))$. We write $\Sigma_{\mathcal{B}}$ for the set of all strategies. A strategy is called *static*, if it always applies an action to the first entity in any state and for all entities of the same type in any state it picks the same action. A static strategy $\tau$ is essentially a function of the form $\sigma : P \to A$, i.e., for an arbitrary $\pi \in \mathit{FPath}_{\mathcal{B}}$, we have $\tau(\pi) = (1, \sigma(\mathit{last}(\pi)_1))$ whenever $\mathit{last}(\pi) \neq \epsilon$.

A strategy $\sigma \in \Sigma_{\mathcal{B}}$ and an initial state $\alpha$ induce a probability measure over the set of runs of BMDP $\mathcal{B}$ in the following way: the basic open sets of $\mathit{Runs}_{\mathcal{B}}$ are of the form $\pi \cdot (\mathit{Actions}_{\mathcal{B}} \times \mathit{States}_{\mathcal{B}})^\omega$, where $\pi \in \mathit{FPath}_{\mathcal{B}}$, and the measure of this open set is equal to $\prod_{i=0}^{|\pi|-1} \sigma(\pi(i))(\pi_{A(i+1)}) \cdot \mathit{Prob}_{\mathcal{B}}(\pi_{S(i)}, \pi_{A(i+1)})(\pi_{S(i+1)})$ if $\pi_{S(0)} = \alpha$ and equal to 0 otherwise. It is a classical result of measure theory that this extends to a unique measure over all Borel subsets of $\mathit{Runs}_{\mathcal{B}}$ and we will denote this measure by $P^{\sigma}_{\mathcal{B},\alpha}$.

Let $f : \mathit{Runs}_{\mathcal{B}} \to \widetilde{\mathbb{R}}_+$ be a function measurable with respect to $P^{\sigma}_{\mathcal{B},\alpha}$. The expected value of $f$ under strategy $\sigma$ when starting at $\alpha$ is defined as $\mathbb{E}^{\sigma}_{\mathcal{B},\alpha}\{f\} = \int_{\mathit{Runs}_{\mathcal{B}}} f \, \mathrm{d}P^{\sigma}_{\mathcal{B},\alpha}$ (which can be $\infty$ even if the probability that the value of $f$ is infinite is 0). The infimum expected value of $f$ in $\mathcal{B}$ when starting at $\alpha$ is defined as $\mathcal{V}_*(\alpha)(f) = \inf_{\sigma \in \Sigma_{\mathcal{B}}} \mathbb{E}^{\sigma}_{\mathcal{B},\alpha}\{f\}$. A strategy, $\widehat{\sigma}$, is said to be optimal if $\mathbb{E}^{\widehat{\sigma}}_{\mathcal{B},\alpha}\{f\} = \mathcal{V}_*(\alpha)(X)$ and $\varepsilon$-optimal if $\mathbb{E}^{\widehat{\sigma}}_{\mathcal{B},\alpha}\{f\} \leq \mathcal{V}_*(\alpha)(f) + \varepsilon$. Note that $\varepsilon$-optimal strategies always exists by definition. We omit the subscript $\mathcal{B}$, e.g., in $\mathit{States}_{\mathcal{B}}$, $\Sigma_{\mathcal{B}}$, etc., when the intended BMDP is clear from the context.

For a given BMDP $\mathcal{B}$ and $N \geq 0$ we define $\mathrm{Total}_N(\pi)$, the cumulative cost of a run $\pi$ after $N$ steps, as $\mathrm{Total}_N(\pi) = \sum_{i=0}^{N-1} \mathit{Cost}(\pi_{S(i)}, \pi_{A(i+1)})$. For a configuration $\alpha \in \mathit{States}$ and a strategy $\sigma \in \Sigma$, let $\mathrm{ETotal}_N(\mathcal{B}, \alpha, \sigma)$ be the *N-step expected total cost* defined as $\mathrm{ETotal}_N(\mathcal{B}, \alpha, \sigma) = \mathbb{E}^{\sigma}_{\mathcal{B},\alpha}\big\{\mathrm{Total}_N\big\}$ and the *expected total cost* be $\mathrm{ETotal}_*(\mathcal{B}, \alpha, \sigma) = \lim_{N \to \infty} \mathrm{ETotal}_N(\mathcal{B}, \alpha, \sigma)$. This last value can potentially be $\infty$. For each starting state $\alpha$, we compute the *optimal expected cost* over all strategies of a BMDP starting at $\alpha$, denoted by $\mathrm{ETotal}_*(\mathcal{B}, \alpha)$, i.e.,

$$\mathrm{ETotal}_*(\mathcal{B}, \alpha) = \inf_{\sigma \in \Sigma_{\mathcal{B}}} \mathrm{ETotal}(\mathcal{B}, \alpha, \sigma).$$

As we are going to prove in Theorem 4.b that, for any $\alpha \in \mathit{States}$, we have

$$\mathrm{ETotal}_*(\mathcal{B}, \alpha) = \sum_{i=1}^{|\alpha|} \mathrm{ETotal}_*(\mathcal{B}, \alpha_i).$$

This justifies focusing on this value for initial states that consist of a single entity only, as we will do in the following section.

## 3 Fixed Point Equations

Following [8], we define here a linear equation system with a minimum operator whose *Least Fixed Point* solution yields the desired optimal values for each type of a BMDP with non-negative costs. This system generalises the Bellman's equations for finite-state MDPs. We use a variable $x_q$ for each unknown $\mathrm{ETotal}_*(\mathcal{B}, q)$ where $q \in P$. Let $\bar{x}$ be the vector of all $x_q$, where$q \in P$. The system has one equation of the form $x_q = F_q(\bar{x})$ for each type $q \in P$, defined as

$$x_q = \min_{a \in A(q)} \Big( c(q,a) + \sum_{\alpha \in P^*} p(q,a)(\alpha) \sum_{i \leq |\alpha|} x_{\alpha_i} \Big) \; . \tag{♠}$$

We denote the system in vector form by $\bar{x} = F(\bar{x})$. Given a BMDP, we can easily construct its associated system in linear time. Let $\bar{c}^* \in \widetilde{\mathbb{R}}^n_{\geq 0}$ denote the $n$-dimensional vector of $\mathrm{ETotal}_*(\mathcal{B}, q)$'s where $n = |P|$. Let us define $\bar{x}^0 = \bar{0}$, $\bar{x}^{k+1} = F^{k+1}(\bar{0}) = F(\bar{x}^k)$, for $k \geq 0$.

**Theorem 4.** *The following hold:*

*(a) The map $F : \widetilde{\mathbb{R}}^n_{\geq 0} \to \widetilde{\mathbb{R}}^n_{\geq 0}$ is monotone and continuous (and so $\bar{0} \leq \bar{x}^k \leq \bar{x}^{k+1}$ for all $k \geq 0$).*
*(b) $\bar{c}^* = F(\bar{c}^*)$.*
*(c) For all $k \geq 0$, $\bar{x}^k \leq \bar{c}^*$.*
*(d) For all $\bar{c}' \in \widetilde{\mathbb{R}}^n_{\geq 0}$, if $\bar{c}' = F(\bar{c}')$, then $\bar{c}^* \leq \bar{c}'$.*
*(e) $\bar{c}^* = \lim_{k \to \infty} \bar{x}^k$.*

*Proof.*

(a) All equations in the system $F(x)$ are minimum of linear functions with non-negative coefficients and constants, and hence monotonicity and continuity are preserved.
(b) It suffices to show that once action $a$ is taken when starting with a single entity $q$ and, as a result, $q$ is replaced by $\alpha$ with probability $p(q,a)(\alpha)$, then the expected total cost is equal to:

$$c(q,a) + \sum_{i \leq |\alpha|} \mathrm{ETotal}_*(\mathcal{B}, \alpha_i) \; . \tag{♣}$$

This is because then the expected total cost of picking action $a$ when at $q$ is just a weighted sum of these expressions with weights $p(q,a)(\alpha)$ for offspring $\alpha$. And finally, to optimise the cost, one would pick an action $a$ with the smallest such expected total cost showing that

$$\text{ETotal}_*(\mathcal{B},q) = \min_{a\in A(q)} \Big(c(q,a) + \sum_{\alpha\in P^*} p(q,a)(\alpha) \sum_{i\le|\alpha|} \text{ETotal}_*(\mathcal{B},\alpha_i)\Big)$$

indeed holds.

Now, to show (♣), consider an $\epsilon$-optimal strategy $\sigma_i$ for a BMDP that starts at $\alpha_i$. It can easily be composed into a strategy $\sigma$ that starts at $\alpha$ just by executing $\sigma_1$ first until all descendants of $\alpha_1$ die out, before moving on to $\sigma_2$, etc. If one of these strategies, $\sigma_i$, never stops executing then, due to the assumption that all costs are positive, the expected total cost when starting with $\alpha_i$ has to be infinite and so has to be the overall cost when starting with $\alpha$ (as all descendants of $\alpha_i$ have to die out before the overall process terminates), so (♣) holds. This shows that $c(q,a) + \sum_{i\le|\alpha|} \text{ETotal}_*(\mathcal{B},x_{\alpha_i})$ can be achieved when starting at $\alpha$. At the same time, we cannot do better because that would imply the existence of a strategy $\sigma'$ for one of the entities $\sigma_j$ with a better cost than its optimal cost $\text{ETotal}_*(\mathcal{B},\alpha_j)$.

(c) Since $\bar{x}^0 = \bar{0} \le \bar{c}^*$ and due to (b), it follows by repeated application of $F$ to both sides of this inequality that $\bar{x}^k \le F(\bar{c}^*) = \bar{c}^*$, for all $k \ge 0$.

(d) Consider any fixed point $\bar{c}'$ of the equation system $F(\bar{x})$. We will prove that $\bar{c}^* \le \bar{c}'$. Let us denote by $\sigma'$ a static strategy that picks for each type an action with the minimum value of operator $F$ in $\bar{c}'$, i.e., for each entity $q$ we choose $\sigma'(q) = \arg\min_{a\in A(q)} \Big(c(q,a) + \sum_{\alpha\in P^*} p(q,a)(\alpha) \sum_{i\le|\alpha|} \bar{c}'_{\alpha_i}\Big)$, where we break ties lexicographically.

We now claim that, for all $k \ge 0$, $\text{ETotal}_k(\mathcal{B},q,\sigma') \le \bar{c}'_q$ holds. For $k=0$, this is trivial as $\text{ETotal}_k(\mathcal{B},q,\sigma') = 0 \le \bar{c}'_q$. For $k > 0$, we have that

$$\begin{aligned}\text{ETotal}_k(\mathcal{B},q,\sigma') &\overset{(1)}{\le} c(q,\sigma'(q)) + \sum_{\alpha\in P^*} p(q,\sigma'(q))(\alpha) \sum_{i\le|\alpha|} \text{ETotal}_{k-1}(\mathcal{B},\alpha_i,\sigma') \\ &\overset{(2)}{\le} c(q,\sigma'(q)) + \sum_{\alpha\in P^*} p(q,\sigma'(q))(\alpha) \sum_{i\le|\alpha|} \bar{c}'_{\alpha_i} \\ &\overset{(3)}{=} \min_{a\in A(q)} \Big(c(q,a) + \sum_{\alpha\in P^*} p(q,a)(\alpha) \sum_{i\le|\alpha|} \bar{c}'_{\alpha_i}\Big) \overset{(4)}{=} \bar{c}'_q\end{aligned}$$

where (1) follows from the fact that after taking action $\sigma'(q)$ first, there are only $k-1$ steps left of the BMDP $\mathcal{B}$ that would need to be distributed among the offspring $\alpha$ of $q$ somehow. Allowing for $k-1$ steps for each of the entities $\alpha_i$ is clearly an overestimate of the actual cost. (2) follows from the inductive assumption. (3) follows from the definition of $\sigma'$. The last equality, (4), follows from the fact that $\bar{c}'$ is a fixed point of $F$.

Finally, for every $q \in P$, from the definition we have $\bar{c}^*_q = \text{ETotal}_*(\mathcal{B},q) \le$

ETotal$_*(\mathcal{B}, q, \sigma') = \lim_{k\to\infty}$ ETotal$_k(\mathcal{B}, q, \sigma')$ and each element of the last sequence was just shown to be $\leq \bar{c}'_q$.

(e) We know that $\bar{x}^* = \lim_{k\to\infty} \bar{x}^k$ exists in $\widetilde{\mathbb{R}}^n_{\geq 0}$ because it is a monotonically non-decreasing sequence (note that some entries may be infinite). In fact we have $\bar{x}^* = \lim_{k\to\infty} F^{k+1}(\bar{0}) = F(\lim_{k\to\infty} F^k(\bar{0}))$, and thus $\bar{x}^*$ is a fixed point of $F$. So from (d) we have $\bar{c}^* \leq \bar{x}^*$. At the same time, due to (c), we have $\bar{x}^k \leq \bar{c}^*$ for all $k \geq 0$, so $\bar{x}^* = \lim_{k\to\infty} \bar{x}^k \leq \bar{c}^*$ and thus $\lim_{k\to\infty} \bar{x}^k = \bar{c}^*$.

□

The following is a simple corollary of Theorem 4.

**Corollary 5.** *In BMDPs, there exists an optimal static control strategy $\sigma^*$.*

*Proof.* It is enough to pick as $\sigma^*$, the strategy $\sigma'$ from Theorem 4.d, for $\bar{c}' = \bar{c}^*$. We showed there that for all $k \geq 0$ and $q \in P$ we have ETotal$_k(\mathcal{B}, q, \sigma^*) \leq \bar{c}'_q$. So ETotal$_*(\mathcal{B}, q, \sigma^*) = \lim_{k\to\infty}$ ETotal$_k(\mathcal{B}, q, \sigma^*) \leq \bar{c}^*_q =$ ETotal$_*(\mathcal{B}, q)$, so in fact ETotal$_*(\mathcal{B}, q, \sigma^*) =$ ETotal$_*(\mathcal{B}, q)$ has to hold as clearly ETotal$_*(\mathcal{B}, q, \sigma^*) \geq$ ETotal$_*(\mathcal{B}, q)$. □

Note that for a BMDPs with a fixed static strategy $\sigma$ (or equivalently BMCs), we have that $F(\bar{x}) = B_\sigma \bar{x} + \bar{c}_\sigma$, for some non-negative matrix $B_\sigma \in \mathbb{R}^{n\times n}_{\geq 0}$, and a positive vector $\bar{c}_\sigma > 0$ consisting of all one step costs $c(q, \sigma(q))$. We will refer to $F$ as $F_\sigma$ in such a case and exploit this fact later in various proofs.

We now show that $\bar{c}^*$ is in fact essentially a unique fixed point of $F$.

**Theorem 6.** *If $F(\bar{x}) = \bar{x}$ and $\bar{x}_q < \infty$ for some $q \in P$ then $\bar{x}_q = \bar{c}^*_q$.*

*Proof.* By Corollary 5, there exists an optimal static strategy, denoted by $\sigma^*$, which yields the finite optimal reward vector $\bar{c}^*$.

We clearly have that $\bar{x} = F(\bar{x}) \leq F_{\sigma^*}(\bar{x})$, because $\sigma^*$ is just one possible pick of actions for each type rather than the minimal one as in (♠). Furthermore,

$$\begin{aligned} F_{\sigma^*}(\bar{x}) &= B_{\sigma^*}\bar{x} + b_{\sigma^*} \\ &\leq B_{\sigma^*}(B_{\sigma^*}\bar{x} + b_{\sigma^*}) + b_{\sigma^*} \\ &= B^2_\sigma \bar{x} + (B_{\sigma^*} + 1) b^*_\sigma \\ &\leq \ldots \leq \lim_{k\to\infty} B^k_{\sigma^*}\bar{x} + \Big(\sum_{k=0}^{\infty} B^k_{\sigma^*}\Big) b_{\sigma^*}. \end{aligned}$$

Note that $\bar{c}^* = (\sum_{k=0}^{\infty} B^k_{\sigma^*}) b_{\sigma^*}$, because

$$\bar{c}^* = \lim_{k\to\infty} F^k(\bar{0}) = \lim_{k\to\infty} F^k_{\sigma^*}(\bar{0}) = \lim_{k\to\infty} \sum_{i=0}^{k} B^i_{\sigma^*} b_{\sigma^*}.$$

Due to Theorem 4.d, we know that $\bar{c}^*_q \leq \bar{x}_q < \infty$, so all entries in the $q$-th row of $B^k_{\sigma^*}$ have to converge to 0 as $k \to \infty$, because otherwise the $q$-th row of $\sum_{k=0}^{\infty} B_{\sigma^*}^k$ would have at least one infinite value and, as a result, the $q$-th position of $\bar{c}^* = (\sum_{k=0}^{\infty} B_{\sigma^*}^k) b_{\sigma^*}$ would also be infinite as all entries of $b_{\sigma^*}$ are positive. Therefore, $\lim_{k\to\infty}(B_{\sigma^*}^k \bar{x})_q = 0$ and so

$$\bar{x}_q \leq (\lim_{k\to\infty} B_{\sigma^*}^k \bar{x})_q + ((\sum_{k=0}^{\infty} B_{\sigma^*}^k) b_{\sigma^*})_q = \bar{c}_q^*.$$

The proof is now complete. $\square$

## 4 Q-learning

We next discuss the applicability of Q-learning to the computation of the fixed point defined in the previous section.

Q-learning [30] is a well-studied model-free RL approach to compute an optimal strategy for discounted rewards. Q-learning computes so-called Q-values for every state-action pair. Intuitively, once Q-learning has converged to the fixed point, $Q(s,a)$ is the optimal reward the agent can get while performing action $a$ after starting at $s$. The Q-values can be initialised arbitrarily, but ideally they should be close to the actual values. Q-learning learns over a number of episodes, each consisting of a sequence of actions with bounded length. An episode can terminate early if a sink-state or another non-productive state is reached. Each episode starts at the designated initial state $s_0$. The Q-learning process moves from state to state of the MDP using one of its available actions and accumulates rewards along the way. Suppose that in the $i$-th step, the process has reached state $s_i$. It then either performs the currently (believed to be) optimal action (so-called *exploitation* option) or, with probability $\epsilon$, picks uniformly at random one of the actions available at $s_i$ (so-called *exploration* option). Either way, if $a_i$, $r_i$, and $s_{i+1}$ are the action picked, reward observed and the state the process moved to, respectively, then the Q-value is updated as follows:

$$Q_{i+1}(s_i, a_i) = (1-\lambda_i) Q_i(s_i, a_i) + \lambda_i (r_i + \gamma \cdot \max_a Q_i(s_{i+1}, a)) \ ,$$

where $\lambda_i \in ]0,1[$ is the learning rate and $\gamma \in ]0,1]$ is the discount factor. Note the model-freeness: this update does not depend on the set of transitions nor their probabilities. For all other pairs $s, a$ we have $Q_{i+1}(s,a) = Q_i(s,a)$, i.e., they are left unchanged. Watkins and Dayan showed the convergence of $Q$-learning [30].

**Theorem 7 (Convergence [30]).** *For $\gamma < 1$, bounded rewards $r_i$ and learning rates $0 \leq \lambda_i < 1$ satisfying:*

$$\sum_{i=0}^{\infty} \lambda_i = \infty \ \mathit{and} \sum_{i=0}^{\infty} \lambda_i^2 < \infty,$$

*we have that $Q_i(s,a) \to Q(s,a)$ as $i \to \infty$ for all $s, a \in S \times A$ almost surely if all $(s,a)$ pairs are visited infinitely often.*

However, in the total reward setting that corresponds to $Q$-learning with discount factor $\gamma = 1$, $Q$-learning may not converge, or converge to incorrect values. However, it is guaranteed to work for finite-state MDPs in the setting of undiscounted total reward with a target sink-state under the assumption that all strategies reach that sink-state almost surely. The assumption that we make instead is that every transition of BMDP incurs a positive cost. This guarantees that a process that does not terminate almost surely generates an expected infinite reward in which case the Q-learning will coverage (or rather diverge) to $\infty$, so our results generalise these existing results for Q-learning.

We adopt the Q-learning algorithm to minimise cost as follows. Each episode starts at the designated initial state $q_0 \in P$. The Q-learning process moves from state to state of the BMDP using one of its available actions and accumulates costs along the way. Suppose that, in the $i$-th step, the process has reached state $\alpha$. It then selects uniformly at random one of the entities of $\alpha$, e.g., the $j$-th one, $\alpha_j$ and either performs the currently (believed to be) optimal action or, with probability $\epsilon$, picks an action uniformly at random among all the actions available for $\alpha_j$. If $c$ and $\beta$ denote the observed cost and entities spawned by this action, respectively, then the Q-value of the pair $\alpha_j$, $a_i$ are updated as follows:

$$Q_{i+1}(\alpha_j, a_i) = (1-\lambda_i)Q_i(\alpha_j, a_i) + \lambda_i\big(c + \sum_{i=1}^{|\beta|} \min_{a \in A(\beta_i)} Q_i(\beta_i, a)\big).$$

and all other Q-values are left unchanged. In the next section we show that Q-learning almost surely converges (diverges) to the optimal finite (respectively, infinite) value of $\bar{c}^*$ almost surely under rather mild conditions.

## 5 Convergence of Q-Learning for BMDPs

We show almost sure convergence of the Q-learning to the optimal values $\bar{c}^*$ in a number of stages. We first focus on the case when all optimal values in $\bar{c}^*$ are finite. In such a case, we show a weak convergence of the expected optimal values for BMCs to the unique fixed-point $\bar{c}^*$, as defined in Sect. 3. To establish this, we show that the expected Q-values are monotonically decreasing (increasing) if we start with Q-values $\kappa\bar{c}^*$ for $\kappa > 1$ ($\kappa < 1$). This convergence from above and below gives us convergence in expectation using the squeeze theorem.

We then establish almost sure convergence to $\bar{c}^*$ by proving a contraction argument, with the extra assumption that the selection of the Q-value to update is done independently at random in each step.

In the next step, we extend this result to BMDPs, first establishing that Q-learning will almost surely converge to the *region* of the Q-values less than or equal to $\bar{c}^*$. We then show that, when considering the pointwise limes inferior values of the sequences of Q-values, there is no point in that region such that every $\varepsilon$-ball around it has a non-zero probability to be represented in the limes inferior. This establishes that $\bar{c}^*$ is the fixed point the Q-values converge against.

Only at the very end, we show that Q-learning also converges (or rather diverges) to the optimal value even if that value happens to be infinite. We then turn to a type with non-finite optimal value and provide an argument for the divergence to $\infty$ of its corresponding Q-value.

We assume that all the Q-values are stored in a vector $Q$ of size $(|P| \cdot |A|)$. We also use $Q(q,a)$ to refer to the entry for type $q \in P$ and action $a \in A(q)$. We introduce the *target for* $Q$ *operator*, $T$, that maps a Q-values vector $Q$ to:

$$T(Q)(q,a) = c(q,a) + \sum_{\alpha \in Q^*} p(q,a)(\alpha) \sum_{i=1}^{|\alpha|} \min_{a_i \in A(\alpha_i)} Q(\alpha_i, a_i) \ .$$

We call $T$ the 'target', because, when the $Q(q,a)$ value is updated, then

$$\mathbb{E}(Q_{i+1}(q,a)) = (1-\lambda_i)Q_i(q,a) + \lambda_i T(Q_i)(q,a)$$

holds, whereas otherwise $Q_{i+1}(q,a) = Q_i(q,a)$.

Thus, when $Q(q,a)$ is selected for update with a chance of $p_{q,a}$, we have that

$$\mathbb{E}(Q_{i+1}(q,a)) = (1-\lambda_i p_{q,a})Q_i(q,a) + \lambda_i p_{q,a} T(Q_i)(q,a) \ . \qquad (\heartsuit)$$

## 5.1 Convergence for BMCs with Finite $\bar{c}^*$

Since BMCs have only one action, we omit mentioning it for ease of notation.

Note that for BMCs, the target for the Q-values is a simple affine function:

$$T(Q)(q) = c(q) + \sum_{\alpha \in P^*} p(q)(\alpha) \sum_{i=1}^{|\alpha|} Q(\alpha_i).$$

And it coincides with operator $F$ as defined in Sect. 3. Therefore, due to Theorem 6, $T(Q)$ has a unique fixed point which is $\bar{c}^*$. Moreover, $T(Q) = BQ + \bar{c}$, where $B$ is a non-negative matrix and $\bar{c}$ is a vector of one step costs $c(q)$, which are all positive.

Naturally, applying $T$ to a non-negative vector $Q$ or multiplying it by $B$ are monotone: $Q \geq Q' \to T(Q) \geq T(Q')$ and $BQ \geq BQ'$. Also, due to the linearity of $T$, $\mathbb{E}(T(Q)) = T(\mathbb{E}(Q))$ holds, where $Q$ is a random vector.

We now start with a lemma describing the behaviour of Q-learning for initial Q-values when they happen to be equal to $\kappa\bar{c}^*$ for some $\kappa \geq 1$.

**Lemma 8.** *Let* $Q_0 = \kappa\bar{c}^*$ *for a scalar factor* $\kappa \geq 1$. *Then the following holds for all* $i \in \mathbb{N}$,

$$\bar{c}^* \leq T(\mathbb{E}(Q_i)) \leq \mathbb{E}(Q_{i+1}) \leq \mathbb{E}(Q_i),$$

*assuming that Q-value to be updated in each step is selected independently at random.*

*Proof.* We show this by induction. For the induction basis ($i = 0$), we have that $\bar{c}^* \leq Q_0$ by definition.

As $\bar{c}^*$ is the fixed-point of $T$, we have $T(\bar{c}^*) = \bar{c}^*$, and the monotonicity of $T$ provides $T(\bar{c}^*) \leq T(Q_0)$. At the same time

$$\begin{aligned} T(Q_0) = T(\kappa\bar{c}^*) &= B\kappa\bar{c}^* + \bar{c} \\ &= \kappa(B\bar{c}^* + \bar{c}) - \kappa\bar{c} + \bar{c} \\ &= \kappa\bar{c}^* - (\kappa - 1)\bar{c} \\ &= Q_0 - (\kappa - 1)\bar{c} \leq Q_0. \end{aligned}$$

This provides $\bar{c}^* \leq T(\mathbb{E}(Q_0)) \leq \mathbb{E}(Q_0)$. Finally, $T(\mathbb{E}(Q_0)) \leq \mathbb{E}(Q_0)$ entails for a learning rate $\lambda_0 \in [0, 1]$ that $T(\mathbb{E}(Q_0)) \leq \mathbb{E}(Q_1) \leq \mathbb{E}(Q_0)$ due to ($\heartsuit$).

For the induction step ($i \mapsto i + 1$), we use the induction hypothesis

$$\bar{c}^* \leq T(\mathbb{E}(Q_i)) \leq \mathbb{E}(Q_{i+1}) \leq \mathbb{E}(Q_i).$$

The monotonicity of $T$ and $\bar{c}^* \leq \mathbb{E}(Q_{i+1}) \leq \mathbb{E}(Q_i)$ imply that $T(\bar{c}^*) \leq T(\mathbb{E}(Q_{i+1})) \leq T(\mathbb{E}(Q_i))$ holds. With $T(\bar{c}^*) = \bar{c}^*$ (from the fixed point equations) and the induction hypothesis, $\bar{c}^* \leq T(\mathbb{E}(Q_{i+1})) \leq \mathbb{E}(Q_{i+1})$ follows.

Using $T(\mathbb{E}(Q_{i+1})) = \mathbb{E}(T(Q_{i+1}))$, this provides $\mathbb{E}(T(Q_{i+1})) \leq \mathbb{E}(Q_{i+1})$, which implies with $\lambda_{i+1} \in [0, 1]$ that

$$T(\mathbb{E}(Q_{i+1})) = \mathbb{E}(T(Q_{i+1})) \leq \mathbb{E}(Q_{i+2}) \leq \mathbb{E}(Q_{i+1})$$

holds, completing the induction step. □

By simply replacing all $\leq$ with $\geq$ in the above proof, we can get the following for all initial Q-values that happen to be $\kappa\bar{c}^*$ where $\kappa \leq 1$:

**Lemma 9.** *Let $Q_0 = \kappa\bar{c}^*$ for a scalar factor $\kappa \in [0, 1]$. Then the following holds for all $i \in \mathbb{N}$, assuming that the Q-value to update in each step is selected independently at random: $\bar{c}^* \geq T(\mathbb{E}(Q_i)) \geq \mathbb{E}(Q_{i+1}) \geq \mathbb{E}(Q_i)$.* □

We now first establish that the distance between $Q$ and $\bar{c}^*$ can be upper bounded by the distance between $Q$ and $T(Q)$ with a fixed linear factor $\mu > 0$.

**Lemma 10.** *There exists a constant $\mu > 0$ such that*

$$\sum_{q \in P} |(Q - T(Q)(q)| \geq \mu \sum_{q \in P} |(Q - \bar{c}^*(q)|$$

*when $Q_0 = \kappa\bar{c}^*$.*

*Proof.* We show this for $\kappa > 1$. The proof for $\kappa < 1$ is similar, and there is nothing to show for $\kappa = 1$.

We first consider the linear programme with a variable for each type with the following constraints for some fixed $\delta > 0$:

$$Q \geq \bar{c}^*, T(Q) \leq Q, \text{and} \sum_{q \in P} Q(q) = \sum_{q \in P} \bar{c}^*(q) + \delta.$$

An example solution to this constraint system is $Q = (1 + \frac{\delta}{\sum_{q \in P} \bar{c}^*(q)})\bar{c}^*$.

We then find a solution minimising the objective $\sum_{q \in P} |(Q - T(Q)(p)|$, noting that all entries are non-negative due to the first constraint. This is expressed by adding $2|P|$ constraints

$$x_q \geq Q(q) - T(Q)(q)$$
$$x_q \geq T(Q)(q) - Q(q)$$

and minimising $\sum_{q \in P} x_q$.

As $\bar{c}^*$ is the only fixed-point of $T$, and $\sum_{q \in P} Q(q) = \sum_{q \in P} \bar{c}^*(q) + \delta$ implies that, for an optimal solution $Q^*$, $Q^* \neq \bar{c}^*$, we have that

$$\sum_{q \in P} |(Q^* - T(Q^*)(q)| > 0.$$

Due to the constraint $Q \geq \bar{c}^*$, we always have $Q = \bar{c}^* + Q_\Delta$ for some $Q_\Delta > \bar{0}$. We can now re-formulate this linear programme to look for $Q_\Delta$ instead of $Q$:

$$Q_\Delta \geq \bar{0},$$
$$BQ_\Delta \leq Q_\Delta, \text{and}$$
$$\sum_{q \in P} Q_\Delta(q) = \delta,$$

with the objective to minimise $\sum_{q \in P} |(Q_\Delta - BQ_\Delta)(q)|$.

The optimal solution $Q^*_\Delta$ to this linear programme gives an optimal value $Q^* = \bar{c}^* + Q^*_\Delta$ for the former and, vice versa, the value $Q^*$ for the former provides an optimal solution $Q^*_\Delta - \bar{c}^*$ for the latter, and these two solutions have the same value in their respective objective function.

Thus, while the former constraint system is convenient to show that the value of the objective function is positive, the latter constraint system is, except for $\sum_{q \in P} Q_\Delta(q) = \delta$, linear. This means that any optimal solution for $\delta = \delta_1$ can be obtained from the optimal solution for $\delta = \delta_2$ just by rescaling it by $\delta_1/\delta_2$. It follows that the optimal value of the objective function is linear in $\delta$, e.g., there exists $\mu > 0$ such that its value is $\mu\delta$. $\square$

We now show that the sequence of Q-values updates converges in expectation to $\bar{c}^*$ when $Q_0 = \kappa\bar{c}^*$.

**Lemma 11.** *Let $Q_0 = \kappa\bar{c}^*$ where $\kappa \geq 0$. Then, assuming that each type-action pair is selected for update with a minimal probability $p_{\min}$ in each step, and that $\sum_{i=0}^{\infty} \lambda_i = \infty$, then $\lim_{i \to \infty} \mathbb{E}(Q_i) = \bar{c}^*$ holds.*

*Proof.* We proof this for $\kappa \geq 1$. A similar proof shows this for any $\kappa \in [0, 1]$. Lemma 8 provides that all $\mathbb{E}(Q_i)$ satisfy the constraints $\mathbb{E}(Q_i) \geq \bar{c}^*$ and $T(\mathbb{E}(Q_i)) \leq \mathbb{E}(Q_i)$.

Let $p_{\min}$ be the smallest probability any Q-value is selected with in each update step. Due to Lemma 10, there is a fixed constant $\mu > 0$ such that

$$\sum_{q \in P} |Q_i(q) - T(Q_i)(q)| \geq \mu \sum_{q \in P} |Q_i(q) - \bar{c}^*(q)| \ .$$

By taking the expected value of both sides and the fact that $\bar{c}^* \leq T(\mathbb{E}(Q_i)) \leq \mathbb{E}(Q_{i+1}) \leq \mathbb{E}(Q_i)$ due to Lemma 8, we get

$$\sum_{q \in P} \mathbb{E}(Q_i)(q) - T(\mathbb{E}(Q_i))(q) \geq \mu \sum_{q \in P} \mathbb{E}(Q_i)(q) - \bar{c}^*(q),$$

then due to ($\heartsuit$) we have

$$\sum_{q \in P} \mathbb{E}(Q_i)(q) - \mathbb{E}(Q_{i+1})(q) \geq \mu p_{\min} \lambda_i \sum_{q \in P} \mathbb{E}(Q_i)(q) - \bar{c}^*(q),$$

and finally just by rearranging these terms we get

$$\sum_{q \in P} \mathbb{E}(Q_{i+1})(q) - \bar{c}^*(q) \leq (1 - \mu p_{\min} \lambda_i) \sum_{q \in P} \mathbb{E}(Q_i)(q) - \bar{c}^*(q) \ .$$

Note that all summands are positive by Lemma 8.

With $\sum_{i=0}^{\infty} \lambda_i = \infty$, we get that $\sum_{i=0}^{\infty} \mu p_{\min} \lambda_i = \infty$, because $p_{\min}$ and $\mu$ are fixed positive values. This implies that $\prod_{i=0}^{\infty}(1 - \mu p_{\min} \lambda_i) = 0$ and so the distance between $\mathbb{E}(Q_i)$ and $\bar{c}^*$ converges to 0. $\square$

Lemma 11 suffices to show convergence of Q-values in expectation.

**Theorem 12.** *When each Q-value is selected for an update with a minimal probability* $p_{\min}$ *in each step, and* $\sum_{i=0}^{\infty} \lambda_i = \infty$*, then* $\lim_{i \to \infty} \mathbb{E}(Q_i) = \bar{c}^*$ *holds for every starting Q-values* $Q_0 \geq \bar{0}$.

*Proof.* We first note that none of the entries of $\bar{c}^*$ can be 0. This implies that there is a scalar factor $\kappa \geq 0$ such that $\bar{0} \leq Q_0 \leq \kappa \bar{c}^*$. As the $Q_i$ are monotone in the entries of $Q_0$, and as the property holds for $Q_0' = \bar{0} = 0 \cdot \bar{c}^*$ and $Q_0'' = \kappa \bar{c}^*$ by Lemma 11, the squeeze theorem implies that it also holds for $Q_0$. $\square$

Convergence of the expected value is a weaker property than expected convergence, which also explains why our assumptions are weaker than in Theorem 7. With the common assumption of sufficiently fast falling learning rates, $\sum_{i=0}^{\infty} {\lambda_i}^2 < \infty$, we will now argue that the pointwise limes inferior of the sequence of Q-values almost surely converges to $\bar{c}^*$. This will later allow us to infer convergence of the actual sequence of Q-values to $\bar{c}^*$.

**Theorem 13.** *When each Q-value is selected for update with a minimal probability* $p_{\min}$ *in each step,*

$$\sum_{i=0}^{\infty} \lambda_i = \infty \text{ and} \sum_{i=0}^{\infty} \lambda_i^2 < \infty,$$

*then* $\lim_{i \to \infty} Q_i = \bar{c}^*$ *holds almost surely for every starting Q-values* $Q_0 \geq \bar{0}$.

*Proof.* We assume for contradiction that, for some $\widehat{Q} \neq \bar{c}^*$, there is a non-zero chance of a sequence $\{Q_i\}_{i\in\mathbb{N}_0}$ such that

- $\|\widehat{Q} - \liminf_{i\to\infty} Q_i\|_\infty < \varepsilon'$ for all $\varepsilon' > 0$, and
- there is a type $q$ such that $\widehat{Q}(q) < T(\widehat{Q})(q)$.

Then there must be an $\varepsilon > 0$ such that $\widehat{Q}(q) + 3\varepsilon < T(\widehat{Q} - 2\varepsilon \cdot \bar{1})(q)$. We fix such an $\varepsilon > 0$.

Now we have the assumption that the probability of $\|\widehat{Q} - \liminf_{n\to\infty} Q_i\|_\infty < \varepsilon$ is positive. Then, in particular, the chance that, at the same time, $\liminf_{i\to\infty} Q_i > \widehat{Q} - \varepsilon \cdot \bar{1}$ *and* $\liminf_{i\to\infty} Q_i < \widehat{Q} + \varepsilon \cdot \bar{1}$, is positive.

Thus, there is a positive chance that the following holds: there exists an $n_\varepsilon$ such that, for all $i > n_\varepsilon$, $Q_i \geq \widehat{Q} - 2\varepsilon \cdot \bar{1}$. This implies

$$T(Q_i)(q) \geq T(\widehat{Q} - 2\varepsilon \cdot \bar{1})(q) > \widehat{Q}(q) + 3\varepsilon.$$

Thus, the expected limit value of $Q_i(q)$ is at least $\widehat{Q}(q) + 3\varepsilon$, for every tail of the update sequence. Now, we can use $\widehat{Q} - 2\varepsilon$ as a bound on the estimation of the updates in $Q$-learning as $Q_i \geq \widehat{Q} - 2\varepsilon \cdot \bar{1}$ holds. At the same time, the variation of the sum of the updates goes to 0 when $\sum i = 0^\infty \lambda_i^2$ is bounded. Therefore, it cannot be that $\liminf_{i\to\infty} Q_i < \widehat{Q} + \varepsilon \cdot \bar{1}$ holds; a contradiction.

We note that if, for a Q-values $Q \geq \bar{0}$, there is a $q \in P$ with $Q(q') < \bar{c}^*(q')$, then there is a $q \in P$ with $Q(q) < T(Q)(q)$ and $Q(q) < \bar{c}^*(q)$. This is because, for the Q-values $Q'$ with $Q'(q) = \min\{Q(q), \bar{c}^*(q)\}$ for all $q \in Q$, $Q' < \bar{c}^*$. Thus, there must be a type $q \in P$ such that $\kappa = \frac{Q'(q)}{\bar{c}^*(q)} < 1$ is minimal, and $Q' \geq \kappa\bar{c}^*$. As we have shown before, $T(\kappa\bar{c}^*) = \kappa\bar{c}^* - (\kappa - 1)\bar{c}$, such that the following holds:

$$T(Q)(q) \geq T(Q')(q) \geq T(\kappa\bar{c}^*)(q) = \kappa\bar{c}^*(q) + (1-\kappa)c(q) > \bar{c}^*(q) = Q(q).$$

Thus, we have that $\liminf_{i\to\infty} Q_i \geq \bar{c}^*$ holds almost surely. With $\lim_{i\to\infty} \mathbb{E}(Q_i) = \bar{c}^*$, it follows that $\lim_{i\to\infty} Q_i = \bar{c}^*$. □

## 5.2 Convergence for BMDPs and Finite $\bar{c}^*$

We start with showing that, for BMDPs, the pointwise limes superior of each sequence is almost surely less than or equal to $\bar{c}^*$. We then proceed to show that the limes inferior of a sequence is almost surely $\bar{c}^*$, which together implies almost sure convergence.

**Lemma 14.** *When each Q-value of BMDP is selected for update with a minimal probability $p_{\min}$ in each step, $\sum_{i=0}^{\infty} \lambda_i = \infty$, $\sum_{i=0}^{\infty} \lambda_i^2 < \infty$, then $\limsup_{i\to\infty} Q_i \leq \bar{c}^*$ holds almost surely for every starting Q-values $Q_0 \geq \bar{0}$.*

*Proof.* To show the property for the limes superior, we fix an optimal static strategy $\sigma^*$ that exists due to Corollary 5.

We define an BMC obtained by replacing each type $q$ in the BMDP with $A(q) = \{a_1, \ldots, a_k\}$, by $k$ types $(q, a_1), \ldots, (q, a_k)$ with one action, where each type $q'$ is replaced by the type-action pair $(q', \sigma^*(q'))$.

It is easy to see that a type $(q, \sigma^*(q))$ for the resulting BMC has the same value as the type $q$ and the type-action pair $(q, \sigma^*(q))$ in the BMDP that we started with.

When identifying these corresponding type-action pairs, we can look at the same sampling for the BMDP and the BMC, leading to sequences $Q_0, Q_1, Q_2, \ldots$ and $Q'_0, Q'_1, Q'_2, \ldots$, respectively, where $Q_0 = Q'_0$.

It is easy to see by induction that $Q_i \leq Q'_i$. Considering that $\{Q'_i\}_{i \in \mathbb{N}}$ almost surely converges to $\bar{c}^*$ by Theorem 13, we obtain our result. □

**Theorem 15.** *When each Q-value of an BMDP is selected for update with a minimal probability $p_{\min}$, $\sum_{i=0}^{\infty} \lambda_i = \infty$, $\sum_{i=0}^{\infty} \lambda_i^2 < \infty$, then $\lim_{i \to \infty} Q_i = \bar{c}^*$ holds almost surely for every starting Q-values $Q_0 \geq \bar{0}$.*

*Proof.* As a first simple corollary from Lemma 14, we get the same result for the limes inferior (as $\liminf \leq \limsup$ must hold).

We now assume for contradiction that, for some vector $\widehat{Q} < \bar{c}^*$, there is a non-zero chance of a sequence $\{Q_i\}_{i \in \mathbb{N}}$ such that $\|\widehat{Q} - \liminf_{n \to \infty} Q_i\|_\infty < \varepsilon'$ for all $\varepsilon' > 0$.

As $\widehat{Q}$ is below the fixed point of $T$, there must be one type-action pair $(q, \sigma^*(q))$ such that $\widehat{Q}(q, \sigma^*(q)) < T(\widehat{Q})(q, \sigma^*(q))$ (cf. the proof of Theorem 13). Moreover, there must be an $\varepsilon > 0$ such that

$$\widehat{Q}(q, \sigma^*(q)) + 3\varepsilon < T(\widehat{Q} + 2\varepsilon \cdot \bar{1})(q, \sigma^*(q)).$$

We fix such an $\varepsilon > 0$.

Now we assume that the probability of $\|\widehat{Q} - \liminf_{n \to \infty} Q_i\|_\infty < \varepsilon$ is positive. Then the chance that, simultaneously, $\liminf_{i \to \infty} Q_i(q, \sigma^*(q)) > \widehat{Q}(q, \sigma^*(q)) - \varepsilon$ and $\liminf_{i \to \infty} Q_i(q, \sigma^*(q)) < \widehat{Q}(q, \sigma^*(q)) + \varepsilon$, is positive.

Thus, there is a positive chance that the following holds: there exists an $n_\varepsilon$ such that, for all $i > n_\varepsilon$ we have $Q_i \geq \widehat{Q} - 2\varepsilon \cdot \bar{1}$. This entails

$$T(Q_i)(q, \sigma^*(q)) \geq T(\widehat{Q} - 2\varepsilon \cdot \bar{1})(q, \sigma^*(q)) > \widehat{Q}(q, \sigma^*(q)) + 3\varepsilon.$$

Thus, the expected limit value of $Q_i(q, \sigma^*(a))$ is at least $\widehat{Q}(q, \sigma^*(a)) + 3\varepsilon$, for every tail of the update sequence. Now, we can use $T(\widehat{Q} - 2\varepsilon \cdot \bar{1})(q, \sigma^*(a))$ as a bound on the estimation of $T(Q)(q, \sigma^*(q))$ during the update of the Q-value of the type-action pair $(q, \sigma^*(q))$. At the same time, the variation of the sum of the updates goes to 0 when $\sum_{i=0}^{\infty} \lambda_i^2$ is bounded. Therefore, it cannot be that $\liminf_{i \to \infty} Q_i(q, \sigma^*(a)) < \widehat{Q}(q, \sigma^*(a)) + \varepsilon$ holds; a contradiction. □

## 5.3 Divergence

We now show divergence of $Q(q)$ to $\infty$ when at least one of the entries of $\bar{c}^*(q)$ is infinite. First due to Theorem 6 and its proof we have that $\bar{c}^* = \sum_{i=0}^{\infty} B^i \bar{c}$ for some non-negative $B$ and positive $\bar{c}$. Therefore $\bar{c}^*$ is monotonic in $B$ for BMCs. Likewise, the value of $\bar{c}^*$ for a BMDP depends only on the cost function and the expected number of successors of each type spawned: Two BMDPs with same cost functions and the expected numbers of successors have the same fixed point $\bar{c}^*$. Thus, if a type $q$ with one action spawns either exactly one $q$ or exactly one $q'$ with a chance of 50% each, or if it spawns 10 successors of type $q$ and another 10 or type $q'$ with a chance of 5%, while dying without offspring with a chance of 95%, both lead to identical matrices $B$ and so the same $\bar{c}^*$ (though this difference may impact the performance of Q-learning).

Naturally, raising the number of expected number of successors of any type for any type-action pair strictly raises $\bar{c}^*$, while lowering it reduces $\bar{c}^*$, and for every set of expected numbers, the value of $\bar{c}^*$ is either finite or infinite.

Let us consider a set of parameters at the fringe of finite vs. infinite $\bar{c}^*$, and let us choose them pointwise not larger than the parameters from the BMC or BMDP under consideration. As the fixed point from Sect. 3 is clearly growing continuously in the parameter values, this set of expected successors leads to a $\bar{c}^*$ which is not finite.

We now look at the family of parameter values that lead to $\alpha \in [0, 1[$ times the expected successors from our chosen parameter at the fringe between finite and infinite values, and refer to it as the $\alpha$-BMDP. Let also $\bar{c}^*_\alpha$ denote the fixed point for the reduced parameters. As the solution to the fixed point grows continuously, so does $\bar{c}^*_\alpha$. Moreover, if $\bar{c}^*_1 = \lim_{\alpha \to 1} \bar{c}^*_\alpha$ was finite, then $\bar{c}^*$ would be finite as well, because then $\bar{c}^*_1 = \bar{c}^*$.

Clearly, for all parameters $\alpha \in [0, 1[$, the Q-values of an $\alpha$-BMC or $\alpha$-BMDP converge against $\bar{c}^*_\alpha$. Thus, the Q-values for the BMC or BMDP we have started with converges against a value, which is at least $\sup_{\alpha \in [0,1[} \bar{c}^*_\alpha$. As this is not a finite value, Q-learning diverges to $\infty$.

## 6 Experimental Results

We implemented the algorithm described in the previous section in the formal reinforcement learning tool MUNGOJERRIE [21], a C++-based tool which reads BMDPs described in an extension of the PRISM language [18]. The tool provides an interface for RL algorithms akin to that of [3] and invokes a linear programming tool (GLOP) [22] to compute the optimal expected total cost based on the optimality equations (♠).

### 6.1 Benchmark Suite

The BMDPs on which we tested Q-learning are listed in Table 1. For each model, the numbers of types in the BMDP, are given. Table 1 also shows the total cost (as computed by the LP solver), which has full access to the BMDP. This is followed by the estimate of the total cost computed by Q-learning and the time taken by learning. The learner has several hyperparameters: $\epsilon$ is the exploration rate, $\alpha$ is the learning rate, and tol is the tolerance for $Q$-values to be considered different when selecting an optimal strategy. Finally, ep-l is the maximum episode length and ep-n is the number of episodes. The last two columns of Table 1 report the values of ep-l and ep-n when they deviate from the default values. All performance data are the averages of three trials with Q-learning. Since costs are undiscounted, the value of a state-action pair computed by Q-learning is a direct estimate of the optimal total cost from that state when taking that action.

**Table 1.** Q-learning results. The default values of the learner hyperparameters are: $\epsilon = 0.1$, $\alpha = 0.1$, tol= 0.01, ep-l= 30, and ep-n= 20000. Times are in seconds.

| Name | Types | Optimal cost | Estimated cost | Time (avg.) | ep-l | ep-n |
|---|---|---|---|---|---|---|
| `cloud1` | 3 | 5 | 5.026 | 0.369 | | |
| `cloud2` | 4 | 5 | 5.016 | 0.369 | | |
| `bacteria1` | 3 | 2.5 | 2.514 | 0.374 | | |
| `bacteria2` | 3 | 1.34831 | 1.413 | 0.387 | | |
| `protein` | 3 | 6 | 5.067 | 0.372 | | |
| `frozenSmall` | 16 | 1.84615 | 1.740 | 2.834 | 100 | |
| `rand68` | 10 | 150.432 | 154.400 | 0.402 | | |
| `rand283` | 9 | 4 | 4 | 0.075 | | 1000 |
| `rand945` | 19 | 212 | 208.177 | 10.756 | 200 | 40000 |
| `rand3242` | 43 | 4 | 4.372 | 5.960 | 100 | |
| `rand6417` | 62 | 10 | 10 | 12.498 | 50 | |

Models `cloud1` and `cloud2` are based on the motivating example given in the introduction. Examples `bacteria1` and `bacteria2` model the population dynamics of a family of two bacteria [28] subject to two treatments. The objective is to determine which treatment results in the minimum expected cost to extinction of the bacteria population. The `protein` example models a stochastic Petri net description [19] corresponding to a protein synthesis example with entities corresponding to active and inactive genes and proteins. The example `frozenSmall` [3] is similar to classical frozen lake example, except that one of the holes result in branching the process in two entities. Entities that fall in the target cell become extinct. The objective is to determine a strategy that results in a minimum number of steps before extinction. Finally, the remaining 5 examples are randomly created BMDP instances.

## 7 Conclusion

We study the total reward optimisation problem for branching decision processes with unknown probability distributions, and give the first reinforcement learning algorithm to compute an optimal policy. Extending Q-learning is hard, even for branching processes, because they lack a central property of the standard convergence proof: as the value range of the Q-table is not a priori bounded for a given starting table $Q_0$, the variation of the disturbance is not bounded.

This looks like a more substantial obstacle than the one Q-learning faces when maximising undiscounted rewards for finite-state MDPs, and it is well known that this defeats Q-learning. So it is quite surprising that we could not only show that Q-learning works for branching processes, but extend these results to branching decision processes, too. Finally, in the previous section, we have demonstrated that our Q-learning algorithm works well on examples of reasonable size even with default hyperparameters, so it is ready to be applied in practice without the need for excessive hyperparameter tuning.

## References

1. Becker, N.: Estimation for discrete time branching processes with application to epidemics. In: Biometrics, pp. 515–522 (1977)
2. Brázdil, T., Kiefer, S.: Stabilization of branching queueing networks. In: 29th International Symposium on Theoretical Aspects of Computer Science (STACS 2012), vol. 14, pp. 507–518 (2012). https://doi.org/10.4230/LIPIcs.STACS.2012.507
3. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym. CoRR abs/1606.01540 (2016)
4. Chen, T., Dräger, K., Kiefer, S.: Model checking stochastic branching processes. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 271–282. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32589-2_26
5. Esparza, J., Gaiser, A., Kiefer, S.: A strongly polynomial algorithm for criticality of branching processes and consistency of stochastic context-free grammars. Inf. Process. Lett. **113**(10–11), 381–385 (2013)
6. Etessami, K., Stewart, A., Yannakakis, M.: Greatest fixed points of probabilistic min/max polynomial equations, and reachability for branching Markov decision processes. Inf. Comput. **261**, 355–382 (2018). https://doi.org/10.1016/j.ic.2018.02.013
7. Etessami, K., Stewart, A., Yannakakis, M.: Polynomial time algorithms for branching Markov decision processes and probabilistic min(max) polynomial bellman equations. Math. Oper. Res. **45**(1), 34–62 (2020). https://doi.org/10.1287/moor.2018.0970
8. Etessami, K., Wojtczak, D., Yannakakis, M.: Recursive stochastic games with positive rewards. Theor. Comput. Sci. **777**, 308–328 (2019). https://doi.org/10.1016/j.tcs.2018.12.018
9. Etessami, K., Yannakakis, M.: Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. J. ACM **56**(1), 1–66 (2009)
10. Etessami, K., Yannakakis, M.: Recursive Markov decision processes and recursive stochastic games. J. ACM **62**(2), 11:1–11:69 (2015). https://doi.org/10.1145/2699431
11. Even-Dar, E., Mansour, Y., Bartlett, P.: Learning rates for q-learning. J. Mach. Learn. Res. **5**(1) (2003)
12. Haccou, P., Haccou, P., Jagers, P., Vatutin, V.: Branching processes: variation, growth, and extinction of populations. No. 5 in Cambridge Studies in Adaptive Dynamics, Cambridge University Press (2005)
13. Harris, T.E.: The Theory of Branching Processes. Springer, Berlin (1963)

14. Heyde, C.C., Seneta, E.: I. J. Bienaymé: Statistical Theory Anticipated. Springer, Heidelberg (1977). https://doi.org/10.1007/978-1-4684-9469-3
15. Jo, K.Y.: Optimal control of service in branching exponential queueing networks. In: 26th IEEE Conference on Decision and Control, vol. 26, pp. 1092–1097. IEEE (1987)
16. Kiefer, S., Wojtczak, D.: On probabilistic parallel programs with process creation and synchronisation. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 296–310. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_28
17. Kolmogorov, A.N., Sevastyanov, B.A.: The calculation of final probabilities for branching random processes. Doklady Akad. Nauk. U.S.S.R. (N.S.) **56**, 783–786 (1947)
18. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
19. Munsky, B., Khammash, M.: The finite state projection algorithm for the solution of the chemical master equation. J. Chem. Phys. **124**(4), 044104+ (2006)
20. Nielsen, L.R., Kristensen, A.R.: Markov decision processes to model livestock systems. In: Plà-Aragonés, L.M. (ed.) Handbook of Operations Research in Agriculture and the Agri-Food Industry. ISORMS, vol. 224, pp. 419–454. Springer, New York (2015). https://doi.org/10.1007/978-1-4939-2483-7_19
21. Perez, M., Somenzi, F., Trivedi, A.: Mungojerrie: formal reinforcement learning (2021). https://plv.colorado.edu/mungojerrie/. University of Colorado Boulder
22. Perron, L., Furnon, V.: Or-tools (version 7.2) (2019). https://developers.google.com/optimization. Google
23. Pliska, S.R.: Optimization of multitype branching processes. Manag. Sci. **23**(2), 117–124 (1976)
24. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (1994)
25. Rao, A., Bauch, C.T.: Classical Galton-Watson branching process and vaccination. Int. J. Pure Appl. Math. **44**(4), 595 (2008)
26. Rothblum, U.G., Whittle, P.: Growth optimality for branching Markov decision chains. Math. Oper. Res. **7**(4), 582–601 (1982)
27. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. MIT Press, Cambridge (2018)
28. Trivedi, A., Wojtczak, D.: Timed branching processes. In: 2010 Seventh International Conference on the Quantitative Evaluation of Systems, pp. 219–228. IEEE (2010)
29. Udom, A.U.: A Markov decision process approach to optimal control of a multi-level hierarchical manpower system. CBN J. Appl. Stat. **4**(2), 31–49 (2013)
30. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992). https://doi.org/10.1007/BF00992698
31. Watson, H.W., Galton, F.: On the probability of the extinction of families. J. Anthrop. Inst. **4**, 138–144 (1874)
32. Wojtczak, D.: Recursive probabilistic models : efficient analysis and implementation. Ph.D. thesis, University of Edinburgh, UK (2009). http://hdl.handle.net/1842/3217

# Software Verification

# Cameleer: A Deductive Verification Tool for OCaml

Mário Pereira(✉) and António Ravara

NOVA LINCS, Nova School of Science and Technology, Lisbon, Portugal
{mjp.pereira,aravara}@fct.unl.pt

**Abstract.** We present Cameleer, an automated deductive verification tool for OCaml. We leverage on the recently proposed GOSPEL (Generic OCaml SPEcification Language) to attach rigorous, yet readable, behavioral specification to OCaml code. The formally-specified program is fed to our toolchain, which translates it into an equivalent one in WhyML, the programming and specification language of the Why3 verification framework. We report on successful case studies conducted in Cameleer.

**Keywords:** Deductive software verification · OCaml · Why3 · GOSPEL

## 1 Introduction

Over the past decades, we have witnessed a tremendous development in the field of deductive software verification [11], the practice of turning the correctness of code into a mathematical statement and then prove it. Interactive proof assistants have evolved from obscure and mysterious tools into *de facto* standards for proving industrial-size projects. On the other end of the spectrum, the so-called *SMT revolution* and the development of reusable intermediate verification infrastructures contributed decisively to the development of practical automated deductive verifiers.

Despite all the advances in deductive verification and proof automation, little attention has been given to the family of *functional languages* [27]. Let us consider, for instance, the OCaml language. It is well suited for verification, given its well-defined semantics, clear syntax, and state-of-the-art type system. Yet, the community still lacks an easy to use framework for the specification and verification of OCaml code. The working programmers must either re-implement their code in a proof-aware language (and then rely on code extraction), or they must turn themselves into interactive frameworks. Cameleer fills the gap, being a tool for the deductive verification of programs written in OCaml, with a clear

This work is partly supported by the HORIZON 2020 Cameleer project (Marie Skłodowska-Curie grant agreement ID:897873) and NOVA LINCS (Ref. UIDB/04516/2020).

focus on proof automation. Cameleer uses the recently proposed GOSPEL [5], a specification language for OCaml. We advocate here the vision of the *specifying programmer*: the person who writes the code should also be able to naturally provide suitable specification. GOSPEL terms are written in a subset of the OCaml language, which makes them more appealing to the regular programmer. Moreover, we believe specification and implementation should co-exist and evolve together, which is exactly the approach followed in Cameleer.

Cameleer takes as input a GOSPEL-annotated OCaml program and translates it into an equivalent counterpart in WhyML, the programming and specification language of the Why3 framework [16]. Why3 is a toolset for the deductive verification of software, clearly oriented towards automated proof. A distinctive feature of Why3 is that it interfaces with several different off-the-shelf theorem provers, namely SMT solvers.

*Contributions.* To the best of our knowledge, Cameleer is the first deductive verification tool for annotated OCaml programs. It handles a realistic subset of the language, and its interaction with the Why3 verification framework greatly increases proof automation. Our set of case studies successfully verified with the Cameleer tool constitutes, by itself, an important contribution towards building a comprehensive body of verified OCaml codebases. Finally, it is worth noting that the original presentation of GOSPEL was limited to the specification of interface files. In the scope of this work, we have extended it to include implementation primitives, such as loop invariants and ghost code (*i.e.*, code that has no computational purpose and is used only to ease specification and proof effort) evolving GOSPEL from an interface specification language into a more mature proof language.

## 2 Illustrative Example – Binary Search

*Higher-Order Implementation.* Fig. 1 presents an implementation of binary search, where the comparison function, `cmp`, is given as an argument to the main function. For the sake of readability, we give the type of arguments and return value of function `binary_search`, but these can be inferred by the OCaml compiler.

The function contract is given after its definition as a GOSPEL annotation, written within comments of the form `(*@ ... *)`. The first line names the returned value. Next, the first precondition establishes that the `cmp` is a total pre order following the OCaml convention: if `x` is smaller than `y`, then `cmp x y < 0`; if `x` is greater than `y`, then `cmp x y > 0`; finally, `cmp x y = 0` if `x` and `y` are equal values[1]. It is worth noting that GOSPEL, hence Cameleer, assumes `cmp` to be a pure function (*i.e.*, a function without any form of side-effects). The second precondition requires the array to be sorted according to the `cmp` relation. Finally, the last two clauses capture the possible outcomes of execution: the regular postcondition (`ensures` clause) states the returned index is within the bounds of `a` and its value is equal to `v`; the exceptional postcondition (`raises`)

[1] For the sake of space, we omit the definition of predicate `is_total_pre_order`.

```
let binary_search (cmp: 'a -> 'a -> int) (a: 'a array) (v: 'a) : int =
  let l = ref 0 in
  let u = ref (length a - 1) in
  let exception Found of int in
  try while !l <= !u do
      (*@ variant   !u - !l *)
      (*@ invariant 0 <= !l && !u < length a *)
      (*@ invariant forall i. 0 <= i < length a -> cmp a.(i) v = 0 ->
            !l <= i <= !u *)
      let m = !l + (!u - !l) / 2 in
      let c = cmp a.(m) v in
      if c < 0 then l := m + 1
      else if c > 0 then u := m - 1
      else raise (Found m)
    done;
    raise Not_found
  with Found i -> i
(*@ i = binary_search cmp a v
      requires is_total_pre_order cmp
      requires forall i j. 0 <= i <= j < length a -> cmp a.(i) a.(j) <= 0
      ensures  0 <= i < length a && compare a.(i) v = 0
      raises   Not_found -> forall i. 0 <= i < length a -> cmp a.(i) v <> 0 *)
```

**Fig. 1.** Binary search implemented as a functor.

states that whenever exception `Not_found` is raised, there is no such index within bounds whose value is equal to `v`. As usual in deductive verification, the presence of the `while` loop requires one to supply a loop invariant. Here, it boils down to the two `invariant` clauses, which state the limits of the search space are always within the bounds of `a` and that for every index `i` for which `a.(i)` is equal to `v`, then `i` must be within the limits of the current search space. We also provide a decreasing measure (`variant`) in order to prove loop termination.

Assuming file `binary_search.ml` contains the program of Fig. 1, starting a proof with Cameleer is as easy as typing `cameleer binary_search.ml` in a terminal. Users are immediately presented with the Why3 IDE, where they can conduct the proof. Twelve verification conditions are generated for `binary_search`: two for loop invariant initialization, four loop invariant preservation (two for each branch of `if..then..else`), two for safety (check division by zero and index in array bounds), two for loop termination (one for each branch), and finally one for each postcondition. All of these are easily discharged by SMT solvers.

*Functor-Based Implementation.* The implementation in Fig. 2 depicts (the skeleton of) an alternative implementation of the binary search routine. Instead of passing the comparison function as an argument of `binary_search`, here the functor `Make` takes as argument a module of type `OrderedType`, which provides a monomorphic comparison function over a type `t`. This is the same approach found in the OCaml standard library, namely in the `Set` and `Map` modules. The

@logic attribute instructs Cameleer that cmp is both a programming and logical function. This is what allows us to provide the axiom about the behavior of cmp.

Other than the call to Ord.cmp, the implementation and specification of binary_search does not change, hence we omit it here. When fed into Cameleer, the functorial implementation generates the *exact same* twelve verification conditions as the higher-order counterpart, all of them easily discharged as well. Thus, the use of a functor does not impose any verification burden, showing the flexibility of Cameleer to handle different idiomatic OCaml programming styles.

```
module type OrderedType = sig
  type t

  val[@logic] cmp: t -> t -> int
  (*@ axiom total_pre_order: is_total_pre_order cmp *)
end

module Make (Ord: OrderedType) = struct
  let binary_search a v =
    ...
    try while !l <= !u do
        ...
        let c = Ord.cmp a.(m) in
        ...
    (*@ i = binary_search a v ... *)
end
```

**Fig. 2.** Binary search implemented as a functor.

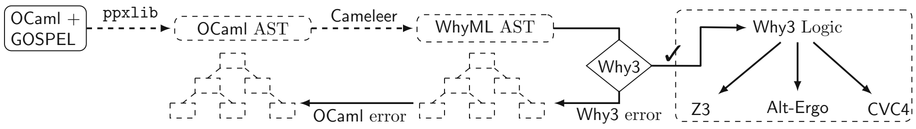


**Fig. 3.** Cameleer verification workflow.

## 3 Implementation

*Cameleer Workflow.* Figure 3 depicts the verification workflow of the Cameleer tool. We use the GOSPEL toolchain[2], in order to parse and manipulate (via the ppxlib library) the abstract syntax tree of the GOSPEL-annotated OCaml program. A dedicated parser and type-checker (extended to handle implementation features) treat GOSPEL special comments and attach the generated specification to nodes in the OCaml AST. Cameleer translates the decorated AST into an

[2] https://github.com/ocaml-gospel/gospel.

equivalent WhyML representation, which is then fed to Why3. The Why3 type-and-effect system might reject the input program, in which case the reported error is propagated back to the level of the original OCaml code. Otherwise, if the translated program fits Why3 requirements, the underlying VCGen computes a set of verification conditions that can then be discharged by different solvers. Throughout all this pipeline, the user only has to write the OCaml code and GOSPEL specification (represented in Fig. 3 as a full-lined box), while every other element is automatically generated (dash-lined boxes). The user never needs to manipulate or even care about the generated WhyML program. In short, the Cameleer user intervenes in the beginning and in the end of the process, *i.e.*, in the initial specifying phase and in the last step, helping Why3 to close the proof. Our development effort currently amounts to 1.8K non-blank lines of OCaml code.

*Translation into WhyML.* The core of Cameleer is a translation from GOSPEL-annotated OCaml code into WhyML. In order to guide our implementation effort, we have defined such a translation as a set of inductive inference rules between the source and target languages [26]. Here, rather than focusing on more fundamental aspects, we give a brief overview of how the translation works in practice.

OCaml and WhyML are both dialects of the ML-family, sharing many syntactic and semantics traits. Hence, translation of OCaml expressions and declarations into WhyML is rather straightforward: GOSPEL annotations are readily translated into WhyML specification, while supported OCaml programming constructions (including ghost code) are easily mapped into semantically-equivalent WhyML constructions. Consider, for instance the following piece of OCaml code:

```
type 'a non_empty_list = { self: 'a list }
(*@ invariant self <> [] *)

let[@ghost] hd (l: 'a non_empty_list) = match l with
  | [] -> assert false
  | x :: _ -> x
(*@ r = hd l
      ensures match l with
              | [] -> false
              | x :: _ -> r = x *)
```

For such case, Cameleer generates the following WhyML program:

```
type non_empty_list 'a = { self: list 'a }
invariant { self <> Nil }

let ghost hd (l: non_empty_list 'a)
  returns { r -> match l with
                 | Nil -> false
                 | Cons x _ -> x = r end }
= match l with
  | Nil -> absurd
  | Cons x _ -> x end
```

Other than the small syntactic differences, the generated WhyML program is identically to the original OCaml one. In particular, the @ghost annotation generates a ghost function in WhyML, while the assert false expression (which is treated in a special way by the OCaml type-checker) is translated into the absurd construction, with the same semantics. Supplied annotations, in this case post-condition and type invariant, are readily mapped into equivalent specification.

The translation of the OCaml module language is more interesting and involved. A WhyML program is a list of modules, a module is a list of top-level declarations, and declarations can be organized within *scopes*, the WhyML unit for namespaces management. However, there is no dedicated syntax for functors on the Why3 side. These are represented, instead, as modules containing only abstract symbols [17]. Thus, when translating OCaml functors into WhyML, we need to be more creative. If we consider, for instance, the Make functor from Fig. 2, Cameleer will generate the following WhyML program:

```
scope Make
  scope Ord
    type t

    val function cmp t t : int
    axiom total_pre_order: is_total_pre_order cmp
  end

  let binary_search a v = ...
end
```

The functor argument Ord is encoded as a nested scope inside Make. This means the binary_search implementation can access any symbol from the Ord namespace, via name qualification (*e.g.*, Ord.t and Ord.cmp).

*Interaction with Why3.* One distinguishing feature of the Why3 architecture is that it can be extended to accommodate new front-end languages [32, Chap. 4]. Building on the devised OCaml to WhyML translation scheme, we use the Why3 API to build an in-memory representation of the WhyML program. We also register OCaml as an admissible input language for Why3, which amounts to instructing Why3 to recognize .ml files as a valid input format and triggering our translation in such case. Following this integration, we can use any Why3 tool, out of the box, to process a .ml file. We are currently using the extract and session tools: the latter to gather statistics about number of generated verification conditions and proof time; the former to erase ghost code.

*Limitations of Using Why3.* The WhyML specification sub-language and GOSPEL are similar. Moreover, they share some fundamental principles, namely the arguments of functions are not aliased by construction and each data structure carries an implicit representation predicate. However, one can use GOSPEL to formally specify OCaml programs which cannot be translated into WhyML. This is evident when it comes to recursive mutable data structures. Consider, for instance, the `cell` type from the `Queue` module of the OCaml standard library[3]:

```
type 'a cell = Nil | Cons of { content: 'a; mutable next: 'a cell }
```

As we attempt to translate such data type, Why3 emits the following error:

```
This field has non-pure type, it cannot be used in a recursive
type definition
```

Recursive mutable data types are beyond the scope of Why3's type-and-effect discipline [14], since these can introduce arbitrary memory aliasing which breaks the *bounded-mutability* principle of Why3 (*i.e.*, all aliases must be statically-known). The solution would be to resort to an axiomatic memory model of OCaml in Why3, or to employ a richer program logic, *e.g.*, Separation Logic [28] or Implicit Dynamic Frames [31]. We describe such an extension as future work (Sect. 6).

## 4 Evaluation

In order to assess the usability and performance of Cameleer, we have put together a test suite of over 1000 lines of OCaml code. The reported case studies are all automatically verified. To build our gallery of verified programs we used a combination of Alt-Ergo 2.4.0, CVC4 1.8, and Z3 4.8.6. Figure 4 summarizes important metrics about our verified case studies: the number of generated verification conditions for each example; the total lines of OCaml code, GOSPEL specification, and lines of ghost (these are also included in the number of OCaml LOC), respectively; the time it takes (in seconds) to replay a proof; and finally, if the proof is immediately discharged, *i.e.*, no extra user effort is required other than writing down suitable specification.

Our test bed includes OCaml implementations issued from realistic and massively used programming libraries: the `List.fold_left` iterator and `Stack` module from the OCaml standard library; the Leftist Heap implementation from `ocaml-containers`[4]; finally, the applicative `Queue` module from `OCamlgraph`[5]. We have used Cameleer to verify programs of different nature. These include: numerical programs (*e.g.*, binary multiplication and fast exponentiation); sorting and searching (*e.g.*, binary search and insertion sort); logical algorithms (conversion of a propositional formula into conjunctive normal form); array scanning (finding duplicate values in an array of integers); small-step iterators; data structures implemented as functors (*e.g.*, Pairing Heaps and Binary Search Trees); historical algorithms (checking a large routine by Turing, Boyer-Moore's majority algorithm, FIND by Hoare, and binary tree same fringe); examples in Rustain Leino's forthcoming textbook "Program Proofs"; and higher-order implementations (height of a binary tree computed in CPS). Both small-step iterators and

[3] https://caml.inria.fr/pub/docs/manual-ocaml/libref/Queue.html.
[4] https://github.com/c-cube/ocaml-containers/blob/master/src/core/CCHeap.ml
[5] https://github.com/backtracking/ocamlgraph/blob/master/src/lib/persistentQueue.ml

| Case study | # VCs | LOC / Spec. / Ghost | Proof time | Immediate |
|---|---|---|---|---|
| Applicative Queue | 23 | 25 / 17 / 4 | 1.26 | ✓ |
| Arithmetic Compiler | 258 | 235 / 44 / 155 | 16.31 | ✗ |
| Binary Multiplication | 12 | 10 / 6 / 0 | 0.69 | ✓ |
| Binary Search | 37 | 62 / 40 / 0 | 1.23 | ✓ |
| Binary Search Trees | 31 | 20 / 26 / 0 | 1.45 | ✗ |
| Checking a Large Routine | 16 | 25 / 15 / 0 | 0.75 | ✓ |
| CNF Conversion | 93 | 113 / 47 / 14 | 2.92 | ✓ |
| Duplicates in an Array | 11 | 10 / 9 / 0 | 0.63 | ✓ |
| Ephemeral Queue | 44 | 40 / 29 / 7 | 1.34 | ✓ |
| Even-odd Test | 6 | 6 / 8 / 0 | 0.55 | ✓ |
| Factorial | 8 | 10 / 9 / 0 | 0.64 | ✓ |
| Fast Exponentiation | 5 | 4 / 5 / 0 | 0.62 | ✓ |
| Fibonacci | 15 | 16 / 15 / 2 | 0.64 | ✓ |
| FIND Algorithm | 6 | 13 / 7 / 0 | 0.57 | ✓ |
| Insertion Sort | 17 | 13 / 34 / 0 | 1.28 | ✓ |
| Integer Square Root | 11 | 8 / 15 / 0 | 0.63 | ✓ |
| Leftist Heap | 161 | 99 / 178 / 11 | 4.33 | ✓ |
| Mjrty | 25 | 33 / 12 / 0 | 2.56 | ✓ |
| OCaml List.fold_left | 28 | 5 / 21 / 0 | 0.79 | ✗ |
| OCaml Stack | 22 | 25 / 27 / 1 | 0.89 | ✓ |
| Pairing Heap | 70 | 65 / 101 / 29 | 2.30 | ✗ |
| Program Proofs | 63 | 93 / 54 / 24 | 1.60 | ✗ |
| Same Fringe | 23 | 22 / 16 / 0 | 0.78 | ✓ |
| Small-step Iterators | 46 | 42 / 52 / 2 | 2.01 | ✗ |
| Tree Height CPS | 4 | 8 / 8 / 0 | 0.80 | ✓ |
| Union Find | 67 | 36 / 29 / 7 | 6.19 | ✓ |

**Fig. 4.** Summary of the case studies verified with the Cameleer tool.

the `list_fold` function use a modular approach to reason about iteration [18]. Our largest case study to date is a toy compiler from arithmetic expressions to a stack machine, while Union Find features the most involved, but very elegant, specification. The former is inspired by the presentation in Nielsons' textbook [25]; the latter follows recently proposed specification techniques [7,12] to achieve fully automatic proofs of correctness and termination.

The runtimes shown in Fig. 4 were measured by averaging over ten runs on a Lenovo Thinkpad X1 Carbon 8th Generation, running Linux Mint 20.1, OCaml 4.11.1, and Why3 1.3.3 (developer version). They show that Cameleer can effectively verify realistic OCaml code in a decent amount of time. Following good practices in deductive verification, Cameleer allows the user to write *ghost code* in order to ease proof and specification. The number of lines of ghost code in Fig. 4 stands for ghost fields in record types, ghost functions, and lemma functions. In particular, the arithmetic compiler example uses lemma functions to prove, by induction, results about semantics preservation. Finally, case studies marked with ✗ required some form of manual interaction in the Why3 IDE [9]. These are very simple proofs by induction (of auxiliary lemmas) and case analysis, in order to better guide SMT solvers.

From our experience developing this gallery of verified programs, we believe the required annotation effort is reasonable, although non-negligible. Some case studies, namely the Heap implementations, feature a considerable amount of lines of GOSPEL specification. However, these are classic definitions (*e.g.*, minimum element) and results (*e.g.*, the root of the Heap is the minimum element), which are easily adapted to any variant of Heap implementation.

## 5 Related Work

*Automated Deductive Verification.* One can cite Why3, F* [1], Dafny [23], and Viper [24] as successful automated deductive verification tools. Formal proofs are conducted in the proof-aware language of these frameworks, and then executable reliable code can be automatically extracted. In the Cameleer project, we chose to develop a verification tool that accepts as input a program written directly in OCaml, instead of a dedicated proof language. This obviates the need to re-write entire OCaml codebases (*e.g.*, libraries), just for the sake of verification.

Regarding tools that tackle the verification of programs written in mainstream languages, one can cite Frama-C [21] (for the C language), VeriFast [20] (C and Java), Nagini [10] (Python), Leon [22] (Scala), Spec# [3] (C#), and Prusti [2] (Rust). Despite the remarkable case studies verified with these tools, programs written in the these languages can quickly degenerate into a nightmare of pointer manipulation and tricky semantics issues. We argue the OCaml language presents a number of features that make it a better target for formal verification.

Finally, language-based approaches offer an alternative path to the verification of software. Liquid Haskell [34] extends standard Haskell types with Liquid Types [29], a form of refinement types [30], in order to prove properties about realistic Haskell programs [33]. In this approach, verification conditions are generated and discharged during type-checking. This is also its major weakness: in order to remain decidable, the expressiveness of the refinement language is hindered. In Cameleer, the use of GOSPEL allows us to provide rich specification to relevant case studies, while still achieving good proof automation results.

*Deductive Verification of OCaml Programs.* Prior to our work, CFML [4] and `coq-of-ocaml` [8] were the only available tools for the deductive verification of OCaml-written code, via translation into the Coq proof language. On one hand, CFML features an embedding of a higher-order Separation Logic in Coq, together with a *characteristic formulae* generator. On the other hand, `coq-of-ocaml` compiles non-mutable OCaml programs to pure Gallina code. These two tools have been successfully applied to the verification of non-trivial case studies, such as the correctness and worst-case amortized complexity bound of cycle detection algorithm [19], as well as part of the Tezos' blockchain protocol[6]. However, they

[6] https://clarus.github.io/coq-of-ocaml/examples/tezos/.

still require a tremendous level of expertise and manual effort from users. Also, no behavioral specification is provided with the OCaml implementation. The user must write specification at the level of the generated code, which breaks our vision that implementation and specification must coexist and evolve together.

The VOCaL project aims at developing a mechanically verified OCaml library [6]. One of the main novelties of this project is the combined use of three different verification tools: Why3, CFML, and Coq. The GOSPEL specification language was developed in the scope of this project, as a tool-agnostic language that could be manipulated by any of the three mentioned frameworks. Up to this point, the three mentioned tools were only using GOSPEL for interface specification, and not as a proof language. We believe the Cameleer approach nicely complements the existing toolchains [13] in the VOCaL ecosystem.

## 6 Conclusions and Future Work

In this paper we presented Cameleer, a tool for automated deductive verification of OCaml programs, with bounded mutability. We use the recently proposed GOSPEL language, which we also extended in the scope of this work, in order to attach formal specification to an OCaml program. Cameleer fulfills a gap in the OCaml community, by providing programmers with a tool to directly specify and verify their implementations. By departing from the interactive-based approach, we believe Cameleer can be an important step towards bringing more OCaml programmers to include formal methods techniques in their daily routines.

The core of Cameleer is a translation from OCaml annotated code into WhyML. The two languages share many common traits (both in their syntax and semantics), so it makes sense to target this intermediate verification language in the first major iteration of Cameleer. We have successfully applied our tool and approach to the verification of several case studies. These include implementations issued from existing libraries, and scale up to data structures implemented as functors and tricky effectful computations. In the future, we intend to apply Cameleer to the verification of even larger case studies.

*What We Do Not Support.* Currently, we target a subset of the OCaml language which roughly corresponds to `caml-light`, with basic support for the module language (including functors). Also, WhyML limits effectful computations to the cases where alias is information statically known, which limits our support for higher-order functions and mutable recursive data structures. Adding support for the objective layer of the OCaml language would require a major extension to the GOSPEL language and a redesign of our translation into WhyML. Nonetheless, Why3 has been used in the past to verify Java-written programs [15], so in principle an encoding of OCaml objects in WhyML is possible.

We do not support some of the more advanced type features in OCaml, namely Generalized Algebraic Data Types (GADTs) and polymorphic variants. One way to support such constructions would to be extend the type system of Why3 itself, which would likely mean a considerable redesign of the WhyML language.

Another possible route is to extend the core of Cameleer with the ability to translate OCaml code into other, richer, verification frameworks.

*Interface with Viper and CFML.* In order to augment the class of OCaml programs we can treat, we plan on extending Cameleer to target the Viper infrastructure and the CFML tool. On one hand, Viper is an intermediate verification language based on Separation Logic but oriented towards SMT-based software verification, allowing one to automatically verify heap-dependent programs. On the other hand, the CFML tool allows one to verify effectful higher-order programs. We plan on extending the CFML translation engine, in order to take source-code level GOSPEL annotations into account. Since it targets the rich proof language and type system of Coq, it can in principle be extended to reason about GADTs and other advanced OCaml features. Even if it relies on an interactive proof assistant, CFML provides a comprehensive tactics library that eases proof effort.

Our ultimate goal is to grow Cameleer to a verification tool that can simultaneously benefit from the best features of different intermediate verification frameworks. Our motto: we want Cameleer to be able to verify parts of OCaml code using Why3, others with Viper, and some very specific functions with CFML.

## References

1. Ahman, D., et al.: Dijkstra monads for free. In: 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 515–529. ACM (2017). https://doi.org/10.1145/3009837.3009878
2. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust Types for Modular Specification and Verification. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360573
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
4. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 418–430 (2011). https://doi.org/10.1145/2034773.2034828
5. Charguéraud, A., Filliâtre, J.-C., Lourenço, C., Pereira, M.: GOSPEL—providing OCaml with a formal specification language. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 484–501. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_29
6. Charguéraud, A., Filliâtre, J.C., Pereira, M., Pottier, F.: VOCAL - A Verified OCAml Library. In: ML Family Workshop (2017). https://hal.inria.fr/hal-01561094
7. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. J. Autom. Reason. **62**(3), 331–365 (2017). https://doi.org/10.1007/s10817-017-9431-7
8. Claret, G.: Program in Coq. (Programmer en Coq). Ph.D. thesis, Paris Diderot University, France (2018). https://tel.archives-ouvertes.fr/tel-01890983

9. Dailler, S., Marché, C., Moy, Y.: Lightweight interactive proving inside an automatic program verifier. In: 4th Workshop on Formal Integrated Development Environment (F-IDE) (2018)
10. Eilers, M., Müller, P.: Nagini: a static verifier for python. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 596–603. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_33
11. Filliâtre, J.C.: Deductive software verification. Int. J. Softw. Tools Technol. Transf. (STTT) **13**(5), 397–403 (2011). https://doi.org/10.1007/s10009-011-0211-0
12. Filliâtre, J.C.: Simpler proofs with decentralized invariants. J. Log. Algebraic Methods Program. (2020, to appear). https://hal.inria.fr/hal-02518570
13. Filliâtre, J.C., et al.: A toolchain to produce verified OCaml libraries. Research report, Université Paris-Saclay (2020). https://hal.archives-ouvertes.fr/hal-01783851
14. Filliâtre, J.C., Gondelman, L., Paskevich, A.: A pragmatic type system for deductive verification. Research report, Université Paris Sud (2016). https://hal.archives-ouvertes.fr/hal-01256434v3
15. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_21
16. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
17. Filliâtre, J.-C., Paskevich, A.: Abstraction and genericity in Why3. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 122–142. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_7
18. Filliâtre, J.-C., Pereira, M.: A modular way to reason about iteration. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 322–336. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_24
19. Guéneau, A., Jourdan, J., Charguéraud, A., Pottier, F.: Formal proof and analysis of an incremental cycle detection algorithm. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, (ITP). LIPIcs, vol. 141, pp. 18:1–18:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.18
20. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
21. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
22. Kuncak, V.: Developing verified software using leon. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 12–15. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_2
23. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20

24. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
25. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science. Springer, Heidelberg (2007). https://doi.org/10.1007/978-1-84628-692-6
26. Pereira, M., Ravara, A.: Cameleer: a deductive verification tool for OCaml. CoRR (2021). https://arxiv.org/abs/2104.11050
27. Régis-Gianas, Y., Pottier, F.: A hoare logic for call-by-value functional programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 305–335. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70594-9_17
28. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS 2002, pp. 55–74. IEEE Computer Society, USA (2002)
29. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Gupta, R., Amarasinghe, S.P. (eds.) 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 159–169. ACM (2008). https://doi.org/10.1145/1375581.1375602
30. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: predicate subtyping in PVS. IEEE Trans. Softw. Eng. **24**(9), 709–720 (1998). https://doi.org/10.1109/32.713327
31. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). https://doi.org/10.1145/2160910.2160911
32. The Why3 Development Team: The Why3 platform, version 1.3.3. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay (2020). http://why3.lri.fr/manual.pdf
33. Vazou, N., Breitner, J., Kunkel, R., Horn, D.V., Hutton, G.: Theorem proving for all: equational reasoning in liquid haskell (functional pearl). In: Wu, N. (ed.) 11th ACM SIGPLAN International Symposium on Haskell, pp. 132–144. ACM (2018). https://doi.org/10.1145/3242744.3242756
34. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Jones, S.L.P.: Refinement types for haskell. In: Jeuring, J., Chakravarty, M.M.T. (eds.) 19th ACM SIGPLAN International Conference on Functional Programming (ICFP), pp. 269–282. ACM (2014). https://doi.org/10.1145/2628136.2628161

# LLMC: Verifying High-Performance Software

Freark I. van der Berg(✉)

Formal Methods and Tools, University of Twente,
Enschede, The Netherlands
f.i.vanderberg@utwente.nl

**Abstract.** Multi-threaded unit tests for high-performance thread-safe data structures typically do not test all behaviour, because only a single scheduling of threads is witnessed per invocation of the unit tests. Model checking such unit tests allows to verify all interleavings of threads. These tests could be written in or compiled to LLVM IR. Existing LLVM IR model checkers like DIVINE and Nidhugg, use an LLVM IR interpreter to determine the next state. This paper introduces LLMC, a multi-core explicit-state model checker of multi-threaded LLVM IR that translates LLVM IR to LLVM IR that is *executed* instead of interpreted. A test suite of 24 tests, stressing data structures, shows that on average LLMC clearly outperforms the state-of-the-art tools DIVINE and Nidhugg.

## 1 Introduction

High-performance software often uses thread-safe data structures to allow multiple threads access to the data, without corrupting it. Unit tests for such data structures typically do not test all behaviour, because the thread scheduler of the run-time environment non-deterministically chooses only a single interleaving. Thus, only a single trace is witnessed each time the unit test is invoked. If we would *model check* [1] these unit tests, we can witness all possible traces by exploring all thread schedules. Because it does not depend on the run-time environment, model checking can become part of a continuous integration pipeline, enabling push-button verification of multi-threaded software.

These thread-safe data structures can be written in or compiled to LLVM IR, the intermediate representation of the LLVM Project [2]. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Many front-ends for LLVM IR exist, for example for C, C++, Java, Ruby, and Rust, potentially allowing an LLVM IR model checker to be usable for many languages.

### 1.1 Related Work

Model checkers that operate on LLVM IR already exist, for example DIVINE, Nidhugg, RCMC and LLBMC. DIVINE [3] is a stateful multi-core model checker of multi-threaded LLVM IR. It has many features such as capturing I/O during model checking, SC and TSO memory models, library support such as `libc` and `libpthread`. Input programs are linked with DIVINE's operating system layer, DiOS, and are interpreted as a whole on the DiVM virtual machine.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 690–703, 2021.
https://doi.org/10.1007/978-3-030-81688-9_32

DIVINE detects memory operations to thread-private memory, by traversing the heap on-the-fly and recognizing if a memory-object is either known only to one thread or to multiple [4]. In the former case, memory operations to that memory-object can be *collapsed*, i.e. joined with the previous instruction.

Nidhugg [5] is a stateless multi-core model checker of multi-threaded LLVM IR that uses an LLVM IR interpreter. It features a sophisticated partial-order reduction, *rfsc* [6], that categorizes traces according to which read reads from which write and traverses only one trace in each category. In practice this reduction is quite powerful. However, Nidhugg comes with a caveat: because Nidhugg is stateless, common prefixes of traces are traversed once per trace instead of once in total. This down-side of a stateless approach becomes more pronounced with longer and more often occurring common traces. Moreover, Nidhugg might not terminate in the presence of infinite loops.

RCMC [7] is also a stateless LLVM IR model checker. During execution within its LLVM IR interpreter, it keeps track of a happens-before graph of all observed memory operations. Using this, RCMC can determine the possible values a read can observe, without simply executing all interleavings of all threads. Unlike Nidhugg, it does not support heap memory and is only released in binary form.

CBMC [8] is a bounded model checker for C and C++ programs, using SMT solving to check for memory safety, exceptions, undefined behaviour and assertions. Loops and recursion are a problem for CBMC when their bound cannot be determined: one needs to set an upper bound on the number of unwindings.

LLBMC [9] is similar to CBMC, using SMT-solving to find bugs, but only for single-threaded C/C++ programs and it operates on LLVM IR.

Other, less related tools include SMACK [10], SeaHorn [11] and KLEE [12].

## 1.2 Contribution

This paper introduces LLMC 0.2, a stateful multi-core model checker of multi-threaded LLVM IR. Instead of using an LLVM IR interpreter like DIVINE, Nidhugg and LLMC 0.1 [13], it transforms input LLVM IR to LLVM IR that implements the DMC API, the next-state interface to the model checker DMC [14]. We call this transformation process LL2DMC and combined with DMC (Fig. 1), it allows for up to three orders of magnitude higher throughput (states/s) than DIVINE. At present, LLMC lacks sophisticated state space reductions, causing state space sizes of roughly two orders of magnitude larger than DIVINE. We compared LLMC to DIVINE and Nidhugg using a test suite covering various data structures. Overall, despite the lack of sophisticated reductions, LLMC is on average an order of magnitude faster than DIVINE and ~3.8x faster than Nidhugg. Additionally, LLMC is able to compute the state spaces of the tests where DIVINE or Nidhugg fail.

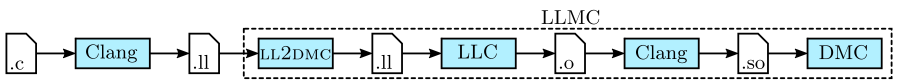


**Fig. 1.** The flow of how an LLVM IR input program is verified in LLMC.

## 2 LLMC: Low-Level Model Checker

This section explains how the transformation process (LL2DMC) transforms the input LLVM IR of a program to LLVM IR that implements the DMC API. LLMC supports LLVM IR compiled from C and C++, by handling a number of builtins (e.g. `__atomic_*` for atomic memory operations), part of `libpthread` (for thread support), `libc` (e.g. for memory allocation) and global constructors.

### 2.1 DMC Model Checker

The model created by LL2DMC is given to DMC to explore. DMC interacts with the model via the DMC API (NEXTSTATE API and DTREE API combined) as illustrated in Fig. 2: after requesting the initial state from the model, DMC continues to request successor states, until the state space has been generated. A state is a vector of 32-bit integers; two states need not be of the same length.

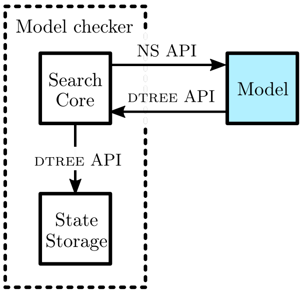


**Fig. 2.** DMC model checker

The states are stored in the concurrent compression tree DTREE [14], allowing lossless compression, fast insertion and duplicate detection of states. When inserted, states are given a unique `StateID`. A `StateID` can be stored in states as well, thus allowing the creation of a DAG of states: a *root-state* and *sub-states*. Additionally, DTREE allows incremental updates to a state, without having the actual contents of the state and it allows partial reconstruction of states. This *delta interface* uses the `StateID` to identify states and can avoid needless copying of entire states, increasing performance. DMC exposes these DTREE features as part of the DMC API [14].

### 2.2 Input Language to LL2DMC: LLVM IR

To understand how LLMC handles input LLVM IR [2], we briefly explain it here. LLVM IR supports control flow by way of basic blocks. Basic blocks are a list of instructions that execute sequentially. The last instruction of a basic block is a terminator instruction, such as a branch (jump) instruction or `return` statement.

LLVM IR uses single static assignment form for register values. To support data flow depending on control flow, $\phi$-nodes exist. These nodes are instructions at the beginning of a basic block that take a value depending on the basic block from which was jumped to the basic block containing the $\phi$-nodes.

### 2.3 Output of LL2DMC: Model Implementing DMC API

The output of LL2DMC is a model that implements the NEXTSTATE API part of the DMC API of the model checker DMC [14]. The NEXTSTATE API requires two interfaces from a model: one to communicate the initial state and one to generate next states, given a state.

The *initial state* of a model generated by LL2DMC is as if one just started the program: registers are unused, global memory is initialized to 0 and a call to the global constructor (`@llvm.global_ctors`) is set up. Global constructors are functions that are called before `main`, which are used to initialize memory and miscellaneous initialization, such that the executable is set up properly before `main` is invoked. Having the initial state in this manner, allows the global constructor to be part of the state space and thus be checked as well.

Starting with the initial state, DMC will keep asking the model to generate the next states for a given state, by invoking the *next-state interface* of the model, until there are no more new states of which to request next states. Given a state, the next-state interface determines the states reachable from that state. In the case of a model generated by LL2DMC, first the global constructors of the modelled program are explored, thus faults in global constructors are detected. When the global constructors are completed, a call to `main` is set up. At this point, the exploration is performed until no new states are visited.

## 2.4 State Space Exploration

This section describes the next-state function and how it is generated from LLVM IR. Figure 3 describes what a state looks like. A state contains information not unlike what an operating system keeps track of [15]. All instructions are mapped to a unique index, such that the `PC` (program counter) uniquely identifies the current position in code. The field `Thread Results` holds the return values of finished threads; the field `#threads` specifies the number of threads in the current state. The remainder of the state constitutes a list of per-thread data.

Each thread has its own `PC` and can independently manipulate it by function calls or branching. `Status` fields are used to indicate whether the thread/program is running, done or failed. Each thread has its own set of `Registers`, the current state of LLVM IR registers. The size of `Registers` is determined by the function requiring the largest number of LLVM IR registers. Function calls manipulate these registers and the list of stack frames described by `Previous frame`.

A `Field` is a `StateID` to a sub-state, as described in Sect. 2.1. The separation into a root-state and sub-states allows sub-states to grow and the state storage component of DMC, DTREE, to compress them using tree compression [14]. It also allows the use of the delta interface: a write to memory can be simply translated

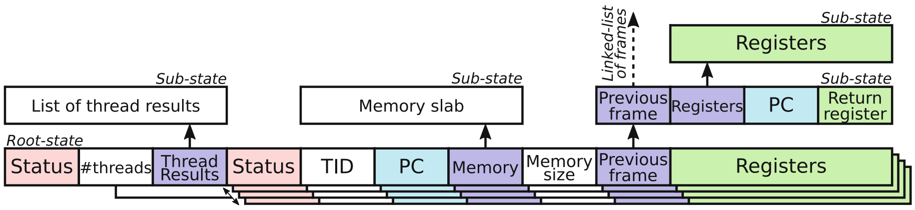


**Fig. 3.** A description of the state used by LLMC.

to a single, efficient call, taking the current `Memory` index, the offset to write to and the new data. The resulting index can be written to `Memory`.

A single LLVM IR instruction in the program is translated to many LLVM IR instructions in the model. We will distinguish LLVM IR registers in the model from registers in the source program by calling the former *model-registers*. In general, a single LLVM IR instruction is translated to a single step with three phases: In the *Preamble* phase, operands to the source LLVM IR instruction are remapped to model-registers and loaded from `Registers` or `Memory`. In the *Action* phase, the source LLVM IR instruction is cloned, with the operands remapped to the LLVM IR model-registers set up during the Preamble phase. In the *Epilogue* phase, if the source LLVM IR instruction assigns a value to a register, the value returned by the cloned instruction is written to `Registers`.

Listing 1 illustrates how a step is performed as part of the next-state function. Multiple steps can be performed as part of the same transition (line 8), as long as the changes are local to the thread (line 4). This is explained in more detail in Sect. 2.5. The `step` function is called for every thread in the state vector.

### 2.4.1 Register Manipulation

Note that the `Registers` are not separated into a sub-state, like `Memory`. We chose this such that simple register manipulating LLVM IR instructions would have no need for an indirection and directly translate to an identical instruction, with its operands mapped such that they are loaded from the `Registers` and the return value of the instruction written back to the corresponding register. This allows us to trivially collapse such instructions, combining the Preamble phases, requiring dependencies only to be loaded once.

### 2.4.2 Memory Instructions

Memory instructions such as loads and stores can be directly mapped to the delta interface, reading or writing only a part of the `Memory` sub-state. There is no distinction between memory allocated on the stack (`alloca`) and on the heap

**Listing 1** In the next-state function, the `step` function is called for each thread.

```
 1 void step(StateVector sv, int threadID)
 2   bool onlyLocal = true; # true while handling commutative instructions
 3   bool emit = false;     # set to true when new state is to be emitted
 4   while(sv.threads[threadID].pc > 0 and onlyLocal)
 5     switch(sv.threads[threadID].pc)
 6       case 0: break; # not running, do nothing
 7       case SomePC: # PC of first instruction of group
 8         # statically collapsed instructions: preamble, action, epilogue
 9         # sv.threads[threadID].pc, onlyLocal and emit may change
10       ...
11   if(emit) MC.insert(sv); # emit new state if needed
```

(`malloc`): both allocate memory by growing the `Memory` sub-state. The returned pointer describes which thread created the memory and the offset within the sub-state. Any thread can write to and read from any such memory location. At present, memory cannot be freed, so `free` has no effect. Because of the tree compression, this has no detrimental effect on memory usage, but does mean LLMC currently does not detect `free`-related bugs.

#### 2.4.3 Branching, Function Calls and Threading

To support control flow in LLMC, the `PC` can be changed to the index assigned to the first instruction in the target basic block. If the target basic block contains $\phi$-nodes, those registers are updated to the value corresponding to the basic block we are branching from.

Function calls set up a new stack frame with the current `Registers`, `PC` and where to write the return value, then pushes it to the linked list of frames pointed to by `Previous frame`. A return from a function pops the top frame from the list of frames, copies the `Registers` into the state vector, updates the `PC` and writes the return value into the right register. There is no bound on the number of frames; the last frame has `Previous frame` set to 0, indicating no next frame.

Threads are created (`pthread_create`) by enlarging the root state with enough space to fit another thread and incrementing `#threads`. When a thread is done, it is marked as such, but not removed from the state vector. This is to retain the memory allocated by a thread. Due to the compression of DTREE, it has little impact on the memory foot print of the state space. The return value from the thread is added to `Thread results`, where it can be read (`pthread_join`).

### 2.5 State Space Reduction

Instructions that only have an effect local to a thread do not change the behaviour of another thread. Such instructions are commutative; their respective ordering is not relevant. Thus, such instructions can be collapsed with the previous or next instruction. For example, instructions that read and write only to registers of a thread are local instructions and do not influence another thread. Branching and function calls are other such commutative instructions.

LLMC collapses commutative instructions statically as well as dynamically. The latter is needed to collapse instructions after conditional control flow, because statically the condition is unknown. On-the-fly, the condition is evaluated, the branch taken and it is determined if the next instruction can be collapsed.

**2.5.1 Thread-Private Memory** LLMC collapses all such commutative instructions, with the important exception of memory operations on memory only accessible to the current thread (memory operations to memory accessible to other threads are never collapsed). This requires knowledge on what memory each thread can access, which LLMC currently does not track. DIVINE implements [4] this by traversing the memory graph in every state, using a run-time type system to identify pointers and how to follow them (edges); each allocation yields a node.

Nidhugg uses a partial-order-reduction [6] that takes into account from which write a value read by a read originates. In this process, memory operations to thread-private memory are indeed collapsed, because a read can read only a single value: the last value written by the thread itself. The current version of LLMC does not feature an on-the-fly state space reduction for memory operations. Instead, we preprocess the input LLVM IR and statically annotate memory operations that cannot be proven to be local to a thread. While this does reduce the state space, because many operations are to stack variables that remain thread-private, it can only approach the on-the-fly reductions of DIVINE and Nidhugg.

## 3 Evaluation

Table 1 shows a feature comparison between the tools mentioned in Sect. 1.1. The table shows that RCMC and CBMC do not support dynamic memory in the presence of multiple threads. This limits their usability for our use case, model checking multi-threaded tests of data structures, since numerous thread-safe data structures use dynamic memory. Furthermore, RCMC, CBMC and LLBMC do not support infinite loops and only have limited support for spin-locks. More complex infinite loops like appending a new node in the Michael-Scott queue [17] using `compare-and-swap` are not supported. Thus, we focus on an experimental comparison between LLMC, DIVINE and Nidhugg on execution time, memory footprint of the state space and scalability across multiple threads, since all three tools support using multiple threads for model checking.

**Table 1.** A feature comparison between the tools mentioned in Sect. 1.1.

| | | | Supported Features | | | | | | | | | Checks | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool | Version | Source | Memory models | Threads | Heap memory | Infinite loops | Global constructors | I/O | Filesystem emulation | `__atomic_*` intrinsics | `atomic {load,store}` | Memory access | `assert()` | Out of Memory | Invalid free | Double free | Memory leaks |
| DIVINE [3] | 4.4.2 (5494190) | LLVM IR | ST[a] | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Nidhugg [5] | 0.2 (45664bc) | LLVM IR | STPWA[a] | ✓ | ✓ | ✓ | $\sim^d$ | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| RCMC [7] | n/a | LLVM IR | (W)RC11 | ✓ | $\sim^b$ | $\sim^c$ | $\sim^d$ | | | ✓ | ✓ | | | | | ✓ | |
| CBMC [8] | 5.10 (ef00f47) | C/C++ | STP[a] | ✓ | $\sim^b$ | $\sim^c$ | $\sim^d$ | | | | | | ✓ | | ✓ | ✓ | ✓ |
| LLBMC [9] | 2013.1 | LLVM IR | n/a | | ✓ | $\sim^c$ | $\sim^d$ | | | | | | ✓ | | ✓ | ✓ | |
| LLMC [13] | 0.1 | LLVM IR | STP[a] | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | | | | |
| LLMC | 0.2 (a732c63) | LLVM IR | S[a] | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | | | | |

[a] Models [16]: **S**) Sequentially consistent; **T**) TSO; **P**) PSO; **W**) POWER; **A**) ARM.
[b] Not supported in combination with threads.
[c] Only trivial spin-locks are supported.
[d] Threads within global constructors not supported.

We ran our experiments on a Dell R930 with 4 E7-8890-v4 CPUs totaling 96 cores and 2 TiB RAM. All sources were compiled using GCC 9.3.0.

### 3.1 Test Suite

We tested the tools using four real-world concurrent LLVM IR data structures, one concurrent algorithm and one protocol. Sources for all tests are available online[1]. We instantiate the tests with various combinations of threads and number of elements inserted, processed or dequeued. All combinations are listed later, in Table 2. These six tests cover different classes of problem types, different shapes of state spaces, and serve to illustrate the strengths and weaknesses of the tools:

- **SortedLinkedList** ● illustrates a concurrency problem where a number of elements are inserted by a number of threads, with a single outcome: all paths converge to one state. Elements can be inserted throughout the chain.
- **LinkedList** ■, similar to SortedLinkedList ●, but with various outcomes, because the list is not sorted. It has high contention on the head of the chain.
- **Prefixsum** ⬢ is a concurrent approach to determine all sums up to any index in an array. It highlights the ability of the model checker to determine thread-private memory, because the two-pass prefixsum algorithm actually partitions the problem into separate per-thread problems that require no communication and one single-threaded part.
- **Hashmap** ♦ illustrates a concurrency problem where a key is inserted using `compare-and-swap`, followed by either atomically storing the value or busy-waiting on the value, if the key already exists (`findOrPut` [18]). The latter involves atomically loading the value until a non-empty value is loaded.
- **MSQ** ▲ is the well-known Michael-Scott queue [17]. It is similar to LinkedList ■, with the addition of dequeue operations, which may return nothing when the queue is empty. The dequeuer can be made *blocking* by calling `dequeue` until it successfully dequeues an element; this is done in ▲2 and ▲4.
- **Philosophers** ▼ is the Dining Philosophers Problem [19], a commonly used protocol to illustrate issues in concurrent resource management. It involves $P$ philosophers and $P$ forks; each philosopher grabs their left fork, then the right, then puts the right fork back, then the left. This is repeated $R$ times. The crux is that each fork is a shared resource for two philosophers. For our tests suite, this illustrates contention on multiple elements in a single array.

These tests highlight the strengths and weaknesses of each tool using real-world data structures and algorithms. The well-known Michael-Scott queue ▲ for example is used in many software packages. They reflect different *kinds* of state spaces: LinkedList ■ focuses on "wide" state spaces, with many end states; SortedLinkedList ● examples state spaces that go wide, but converge into a single end state; Prefixsum ⬢ highlights the model-checker's ability to detect thread-local memory: model checkers that can detect this have a narrow state space, otherwise a model checker will explore all interleavings.

[1] https://github.com/bergfi/llmc/tree/cav2021/tests/performance.

## 3.2 Observations and Considerations

For each model, we verified that all expected end states were reachable. For example for ■1, we manually verified that all $8!/(4!4!) = 70$ possible outcomes of the linked list were generated.

We witnessed DIVINE returning varying state space sizes across different runs on the same test when using multiple threads, indicating a concurrency problem. It also occasionally crashed, most often when using 192 threads. Even though this indicates the answers DIVINE gives might not be correct, we opted to include the results, assuming they would at least provide an indication of the performance.

Furthermore, we did run RCMC on a number of tests. RCMC often runs out of memory before crashing; likely the result of an infinite loop. For even some small tests, it could not finish within 100x the time other tools needed.

## 3.3 Experimental Results

Figure 4 shows the results of LLMC compared to DIVINE on state space exploration time (4a) and Nidhugg on wall-clock time (4b) when applied to the models from Table 2. These graphs indicate relative performance: the uppermost (blue) line for example indicates the line where LLMC is 100x faster. Figure 4c compares LLMC (lower data points) and DIVINE (upper data points) on the memory compression of the state spaces they generate. Figure 4d compares LLMC (upper data points) and DIVINE (lower data points) on the throughput of states per second.

### 3.3.1 LLMC vs DIVINE

Looking at the results in Fig. 4a, we see that LLMC outperforms DIVINE by at least 5x in all test cases except Prefixsum ⬢ and two SortedLinkedList ● tests. LLMC suffers in the Prefixsum ⬢ tests because of the lack of dynamic thread-private memory detection. This results in significantly larger state spaces, up to three orders of magnitude for ⬢4, as seen in Fig. 4c.

Comparing the sorted ● and non-sorted ■ linked list cases, we notice LLMC is able to outperform DIVINE in the non-sorted cases by higher factors than the sorted cases. This difference can be explained by that the two tools generate more similarly sized state spaces for non-sorted ■ cases, but not for sorted ● cases. For example, LLMC generates ∼14.4x more states than DIVINE for ●4, but only ∼2.2x more for ■4. This highlights LLMC is lacking a reduction technique, which works for DIVINE in the sorted cases, but not as well for the non-sorted cases.

For the two Hashmap ♦ cases that both tools completed, LLMC outperforms DIVINE by 8.4x and 157x. Since the hash map is a single global memory object all threads can access, LLMC does not have the disadvantage of lacking a dynamic thread-private memory reduction. DIVINE crashed for the two other ♦ test cases.

DIVINE is unable to complete two of the four Michael-Scott queue ▲ tests, crashing out, the others are verified 86x and 272x faster by LLMC than by DIVINE.

As the complexity of the Philosopher ▼ test cases increases, LLMC increasingly outperforms DIVINE. The two tools generate similarly sized state state spaces,

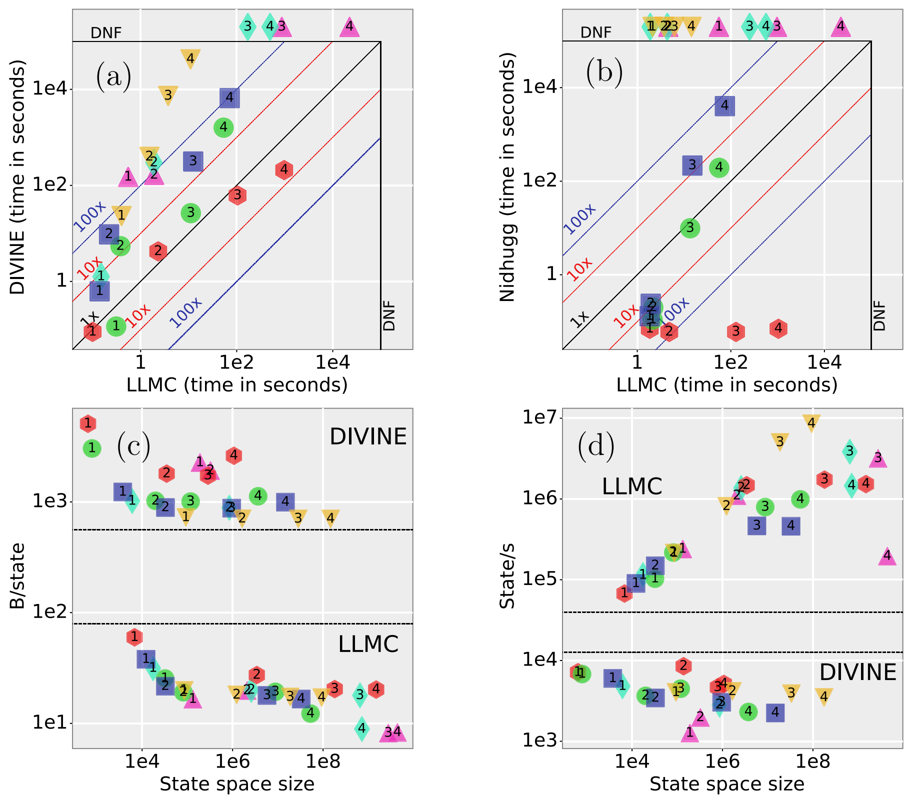


**Fig. 4.** All experimental results, see Table 2 for a legend. Results above the DNF line mean the tool on the y-axis Did Not Finish, not supporting the test.

because the high contention leaves relatively few memory instructions to be collapsed by DIVINE's reduction, thus levelling the playing field.

In summary, LLMC is able to outperform DIVINE in most of the test cases, mostly between 10x–100x faster, with an outlier as high as 2450x faster (▼4). This highlights the performance difference, as on average LLMC visits ∼1.4M

**Table 2.** The six tests with various combinations of number of threads and elements, totaling 24 input programs. MSQ ▲ configurations describe a combination of **E**nqueuers and (**[B]**locking) **D**equeuers in parallel (||) and sequential (;).

| SortedLinkedList | | | LinkedList | | | Prefixsum | | | Hashmap | | | MSQ | | Philosophers | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Threads | Elements | | Thrds | Elems | | Thrds | Elems | | Thrds | KV-pairs | | Configuration | | *P* | *R* |
| ●1 | 2 | 8 | ■1 | 2 | 8 | ⬢1 | 2 | 80 | ◆1 | 3 | 9 | ▲1 | E\|\|E\|\|D\|\|D | ▼1 | 4 | 2 |
| ●2 | 3 | 6 | ■2 | 3 | 6 | ⬢2 | 4 | 80 | ◆2 | 4 | 12 | ▲2 | (E\|\|E\|\|E);(D\|\|D\|\|D) [B] | ▼2 | 4 | 4 |
| ●3 | 3 | 9 | ■3 | 3 | 9 | ⬢3 | 6 | 60 | ◆3 | 4 | 16 | ▲3 | E\|\|E\|\|E\|\|D\|\|D\|\|D | ▼3 | 4 | 8 |
| ●4 | 4 | 8 | ■4 | 4 | 8 | ⬢4 | 6 | 90 | ◆4 | 6 | 12 | ▲4 | E\|\|E\|\|E\|\|D\|\|D [B] | ▼4 | 4 | 12 |

states per second (∼8.5M states/s for ▼4), where DIVINE visits ∼4k states per second (Fig. 4d).

### 3.3.2 LLMC vs Nidhugg

Moving on to Fig. 4b, we notice Nidhugg is unable to complete any of the Michael-Scott queue ▲, Hashmap ◆ or Philosopher ▼ test cases. This is because Nidhugg supports neither the `__atomic_*` instructions needed for the Michael-Scott queue ▲ nor the spin-lock used in the Hashmap ◆ and Philosopher ▼ tests. We tried Nidhugg's transformation capabilities to transform the spin-lock to an assume statement, thus limiting the traces traversed to the ones where the condition of the spin-lock holds, but the generated LLVM IR was invalid and could not be used. Additionally, we tried an experimental version (7b8be8a) with a changelog containing potential fixes to no avail.

We see that Nidhugg outperforms LLMC in the Prefixsum ⬢ test cases consistently by multiple orders of magnitude: Nidhugg traverses only a *single* trace for each of these test cases. This highlights the strength of Nidhugg in its ability to conclude that each read can only read a single value. Without this technique, LLMC needs to exhaustively go through all interleavings of the threads.

For the linked list, sorted ● and non-sorted ■, we see that as the cases get bigger, LLMC is able to outperform Nidhugg. This highlights the disadvantage of stateless model checking: bigger state spaces tend to cause more common prefixes of paths, which causes more work for stateless model checking.

### 3.3.3 Scalability

Figure 5 shows the results for various number of threads for SortedLinkedList3.9 ③, chosen for the performance similarity of the three tools. The graph shown is typical: other test expose similar patterns as the one we highlight here. DIVINE does not scale well in the number of threads: its peak performance lies typically around 4 or 8 threads, confirmed by the DIVINE developers[2]. Nidhugg expectedly does scale very well, as threads just execute a specific trace, with hardly and communication. LLMC shows some scalability, but a ∼4x improvement using 192 threads leaves a lot of room for improvement[3].

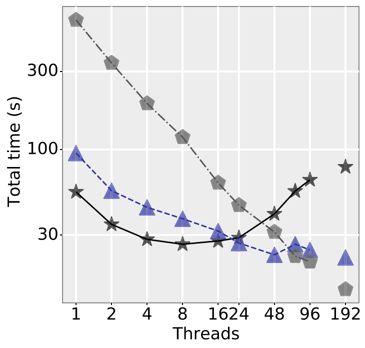


**Fig. 5.** Scalability comparison of DIVINE ★, LLMC ▲, Nidhugg ⬟.

[2] https://divine.fi.muni.cz/trac/ticket/44.
[3] https://github.com/bergfi/dmc/issues/1.

### 3.3.4 DMC and Dtree

We highlight one aspect of the performance of LLMC: the underlying model checker DMC and its storage component DTREE [14]. In Figure 4c, we notice that although LLMC on average generates state spaces of an order of magnitude larger compared to DIVINE, it uses two orders of magnitude less memory per state, due to DTREE. Furthermore, DTREE allows to apply a delta to a state without reconstructing the entire state. Since states are typically ∼2kiB in these tests, this significantly avoids copying memory and increases performance.

## 4 Conclusion

We have introduced LLMC 0.2[4], the multi-threaded low-level model checker that model checks software via LLVM IR. It translates the input LLVM IR into a model LLVM IR that implements the DMC API, the API of the high-performance model checker DMC. This allows LLMC to *execute* the model's next-state function, instead of *interpreting* the input LLVM IR, like DIVINE and Nidhugg. We compared LLMC to these tools using a test suite of 24 tests, covering various data structures. LLMC outperforms DIVINE and Nidhugg up to three orders of magnitude, while other tests have shown areas for improvement. Averaging the results of all completed tests, LLMC is an order of magnitude faster than DIVINE and ∼3.4x faster than Nidhugg. DIVINE and Nidhugg are unable to complete 4 and 12 tests, respectively, due to crashing or not supporting infinite loops or `__atomic_*` library calls.

*Future Work.* LLMC will benefit most from a state space reduction technique that collapses memory instructions to thread-private memory. We aim to integrate this as part of a memory emulation layer that also adds support for relaxed memory models. Even without the dynamic reduction technique, the results show that LLMC in its current form is a high performing tool to model check software.

## References

1. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Lattner, C.: LLVM: an infrastructure for multi-stage optimization. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL, December 2002. http://llvm.cs.uiuc.edu
3. Baranová, Z., et al.: Model checking of C and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
4. Rockai, P., Still, V., Cerná, I., Barnat, J.: DiVM: model checking with LLVM and graph memory. J. Syst. Softw. **143**, 1–13 (2018). https://doi.org/10.1016/j.jss.2018.04.026

[4] https://github.com/bergfi/llmc.

5. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
6. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proceedings of ACM Program. Lang. **3**(dOOPSLA), 150:1–150:29 (2019). https://doi.org/10.1145/3360576
7. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. **2**(POPL), 17:1–17:32 (2018). https://doi.org/10.1145/3158105
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
9. Falke, S., Merz, F., Sinz, C.: The bounded model checker LLBMC. In: Denney, E., Bultan, T., Zeller, A. (eds.) 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, 11–15 November 2013, pp. 706–709. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693138
10. Carter, M., He, S., Whitaker, J., Rakamaric, Z., Emmi, M.: SMACK software verification toolchain. In: Visser, W., Williams, L. (eds.) Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE) Companion. ACM, pp. 589–592 (2016)
11. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
12. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 209–224. USENIX Association (2008)
13. van der Berg, F.I.: Model checking LLVM IR using LTSmin: using relaxed memory model semantics, December 2013. http://essay.utwente.nl/65059/
14. van der Berg, F.I.: Recursive variable-length state compression for multi-core software model checking. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM 2021. LNCS, vol. 12673, pp. 340–357. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_21
15. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 10th edn. Wiley (2018). http://os-book.com/OS10/index.html
16. Gharachorloo, K.: Memory consistency models for shared-memory multiprocessors. Stanford, CA, USA, Technical report (1995). https://doi.org/10.5555/891506
17. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Burns, J.E., Moses, Y. (eds) PODC, pp. 267–275. ACM (1996)
18. van der Berg, F.I., van de Pol, J.: Concurrent chaining hash maps for software model checking. In: Barrett, C., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design (FMCAD), ser. Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 46–54. IEEE, USA, October 2019. https://doi.org/10.23919/FMCAD.2019.8894279
19. Dijkstra, E.W.: Cooperating Sequential Processes, pp. 65–138. Springer, New York (2002). https://doi.org/10.1007/978-1-4757-3472-0_2

# Formally Validating a Practical Verification Condition Generator

Gaurav Parthasarathy[1(✉)], Peter Müller[1], and Alexander J. Summers[2]

[1] Department of Computer Science, ETH Zurich, Zurich, Switzerland
{gaurav.parthasarathy, peter.mueller}@inf.ethz.ch

[2] University of British Columbia, Vancouver, Canada
alex.summers@ubc.ca

**Abstract.** A program verifier produces reliable results only if both the *logic* used to justify the program's correctness is sound, and the *implementation* of the program verifier is itself correct. Whereas it is common to formally prove soundness of the logic, the implementation of a verifier typically remains unverified. Bugs in verifier implementations may compromise the trustworthiness of successful verification results. Since program verifiers used in practice are complex, evolving software systems, it is generally not feasible to formally verify their implementation.

In this paper, we present an alternative approach: we *validate successful runs* of the widely-used Boogie verifier by producing a *certificate* which proves correctness of the obtained verification result. Boogie performs a complex series of program translations before ultimately generating a verification condition whose validity should imply the correctness of the input program. We show how to certify three of Boogie's core transformation phases: the elimination of cyclic control flow paths, the (SSA-like) replacement of assignments by assumptions using fresh variables (passification), and the final generation of verification conditions. Similar translations are employed by other verifiers. Our implementation produces certificates in Isabelle, based on a novel formalisation of the Boogie language.

## 1 Introduction

Program verifiers are tools which attempt to prove the correctness of an implementation with respect to its specification. A successful verification attempt is, however, only meaningful if both the *logic* used to justify the program's correctness is sound, and the *implementation* of the program verifier is itself correct. It is common to formally prove soundness of the logic, but the implementations of program verifiers typically remain unverified. As is standard for complex software systems, bugs in verifier implementations can and do arise, potentially raising doubts as to the trustworthiness of successful verification results.

One way to close this gap is to prove a verifier's implementation correct. However, such a *once-and-for-all* approach faces serious challenges. Verifying an existing implementation bottom-up is not practically feasible because such implementations tend to be large and complex (for instance, the Boogie verifier [29] consists of over 30K lines of imperative C# code), use a variety of libraries, and are typically written in efficient mainstream programming languages which themselves lack a formalisation. Alternatively, one could develop a verifier that is correct by construction. However, this approach requires the verifier to be (re-)implemented in an interactive theorem prover (ITP) such as Coq [14] or Isabelle [24]. This precludes the free choice of implementation language and paradigm, exploitation of concurrency, and possibility of tight integration with standard compilers and IDEs, which is often desirable for program verifiers [4,5,13,26]. Both verification approaches substantially impede software maintenance, which is problematic since verifiers are often rapidly-evolving software projects (for instance, the Boogie repository [1] contains more than 5000 commits).

To address these challenges, in this work we employ a different approach. Instead of verifying the implementation once and for all, we *validate specific runs* of the verifier by automatically producing a *certificate* which proves the correctness of the obtained verification result. Our certificate generation formally relates the input and output of the verifier, but does so largely independently of its implementation, which can freely employ complex languages, algorithms, or optimisations. Our certificates are formal proofs in Isabelle, and so checkable by an independent trusted tool; their guarantees for a certified run of the verifier are as strong as those provided by a (hypothetical) verified verifier.

We apply our novel verifier validation approach to the widely-used Boogie verifier, which verifies programs written in the intermediate verification language Boogie. The Boogie verifier is a *verification condition generator*: it verifies programs by generating a verification condition (VC), whose validity is then discharged by an SMT solver. Certifying a verifier run requires proving that validity of the VC implies the correctness of the input program. Certification of the validity-checking of the VC is an orthogonal concern; our results can be combined with work in that area [11,15,19] to obtain end-to-end guarantees.

Like many automatic verifiers, Boogie is a *translational verifier*: it performs a sequence of substantial Boogie-to-Boogie translations (*phases*), simplifying the task and output of the final efficient VC computation [6,18]. The key challenges in certifying runs of the Boogie tool are to certify each of these phases, including final VC generation. In particular, we present novel techniques for making the following three key phases (and many smaller ones) of Boogie's tool chain certifying:

1. The elimination of loops (more precisely, cycles in the CFG) by reducing the correctness of loops to checking loop invariants *(CFG-to-DAG phase)*
2. The replacement of assignments by (SSA-style) introduction of fresh variables and suitable **assume** statements *(passification phase)*

3. The final generation of the VC, which includes the erasure and logical encoding of Boogie's polymorphic type system [33] *(VC phase)*.

The certification of such verifier phases is related to existing work on compiler verification [34] and validation [8,41,42]. However, the translations and the certified property we tackle here are fundamentally different from those in compilers. Compilers typically require that each execution of the target program corresponds to an execution of the source program. In contrast, the encoding of a program in a translational verifier typically has intentionally more executions (for instance, allows more non-determinism). Moreover, translational verifiers need to handle features not present in standard programming languages such as **`assume`** statements and background theories. Prior work on validating such verifier phases has been limited in the supported language and extent of the formal guarantee; we discuss comparisons in detail in Sect. 8.

**Contributions.** Our paper makes the following technical contributions.

1. The first formal semantics for a significant subset of Boogie (including axioms, polymorphism, type constructors), mechanised in Isabelle.
2. A validation technique for two core program-to-program translations occurring in verifiers (CFG-to-DAG and passification).
3. A validation technique for the VC phase, handling polymorphism erasure and Boogie's type system encoding [31], for which no prior formal proof exists.
4. A version of the Boogie implementation that produces certificates for a significant subset of Boogie.

Making the Boogie verifier certifying is an important result, reducing the trusted code base for a wide variety of verification tools implemented via encodings into Boogie, e.g. Dafny [31], VCC [13], Corral [28], and Viper [35]. Moreover, the technical approach we present here can in future be applied to the certification of the translations performed by these tools, and those based on comparable intermediate verification languages such as Frama-C [26] and Krakatoa [17] based on Why3 [16] and Prusti [4] and VerCors [10] based on Viper [35].

*Outline.* Section 2 explains at a high-level, how our validation approach is structured for the different phases. Section 3 introduces a formal semantics for Boogie. Sections 4, 5 and 6 present our validation of the CFG-to-DAG, passification, and VC phases, respectively. Section 7 evaluates our certificate-producing version of Boogie. Section 8 discusses related work. Section 9 concludes. Further details are available in our accompanying technical report (hereafter, TR) [37].

## 2 Approach

A Boogie program consists of a set of procedures, each with a specification and a procedure body in the form of a (reducible) control-flow-graph (CFG), whose blocks contain basic commands; we present the formal details in the next section. Boogie verifies each procedure modularly, desugaring procedure calls according

$$\begin{tikzcd}
\boxed{G_1} \arrow[r, "\text{CFG-to-DAG}"] & \boxed{G_2} \arrow[l, dotted, bend left, "\mathsf{ver}(G_2) \models \mathsf{ver}(G_1)"] \arrow[r, "\text{Passification}"] & \boxed{G_3} \arrow[l, dotted, bend left, "\mathsf{ver}(G_3) \models \mathsf{ver}(G_2)"] \arrow[r, "\text{VC Phase}"] & \boxed{\mathrm{VC}} \arrow[l, dotted, bend left, "\mathsf{valid}(\mathrm{VC}) \models \mathsf{ver}(G_3)"] \arrow[lll, dashed, bend left=50, "\mathsf{valid}(\mathrm{VC}) \models \mathsf{ver}(G_1)"]
\end{tikzcd}$$

**Fig. 1.** Key phases of verification in Boogie and their certification. The solid edges show Boogie's transformations on a procedure body; each node $G_i$ represents a control-flow-graph. Our final certificate (dashed edge) is constructed by formally linking the three phase certificates represented by the dotted edges. Each of the three phase certificates also incorporate extra smaller transformations that we do not show here.

to their specifications. Verification is implemented via a series of phases: program-to-program translations and a final computation of a VC to be checked by an SMT solver. Our goal is to formally certify (per run of Boogie) that validity of this VC implies the correctness of the original procedure.

To keep the complexity of certificates manageable, our technical approach is *modular* in three dimensions: decomposing our formal goal per *procedure* in the Boogie program, per *phase* of the Boogie verification, and per *block* in the CFG of each procedure. This modularity makes the full automation of our certification proofs in Isabelle practical. In the following, we give a high-level overview of this modular structure; the details are presented in subsequent sections.

*Procedure Decomposition.* Boogie has no notion of a main program or an overall program execution. A Boogie program is correct if each of its procedures is individually correct (that is, the procedure body has no failing traces, as we make precise in the next section). Boogie computes a separate VC for each procedure, and we correspondingly validate the verification of each procedure separately.

*Phase Decomposition.* We break our overall validation efforts down into per-phase sub-problems. In this paper, we focus on the following three most substantial and technically-challenging of these sequential phases, illustrated in Fig. 1. (1) The *CFG-to-DAG phase* translates a (possibly-cyclic) CFG to an acyclic CFG (*cf.* Sect. 4). This phase substantially alters the CFG structure, cutting loops using annotated loop invariants to over-approximate their executions. (2) The *passification phase* eliminates imperative updates by transforming the code into static single assignment (SSA) form and then replacing assignments with *constraints* on variable versions (*cf.* Sect. 5). Both of these phases introduce extra non-determinism and **assume** statements (which, if implemented incorrectly could make verification unsound by masking errors in the program). (3) The final *VC phase* translates the acyclic, passified CFG to a verification condition that, in addition to capturing the weakest precondition, encodes away Boogie's polymorphic type system [33].

We construct certificates for each of these key phases separately (depicted by the blue dotted lines in Fig. 1). For each phase, we certify that *if* the target of the translation phase is correct (a correct Boogie program for the first two phases; a valid VC for the VC phase) then the source (program) of the phase is correct. This modular approach lets us focus the proof strategy for each phase on its conceptually-relevant concerns, and provides robustness against *changes* to the verifier since at most the certification of the changed phases may need adjustment. Logically, our per-phase certificates are finally glued together to guarantee the analogous end-to-end property for the entire pipeline, depicted by the green dashed edge in Fig. 1. For our certificates, we import the input and output programs (and VC) of each key phase from Boogie into Isabelle; we do not reimplement any of Boogie's phases inside Isabelle.

The certificates of the key phases also incorporate various smaller transformations between the key phases, such as peephole optimisation. Our work also validates these smaller transformations, but we focus the presentation on the key phases in this paper. Boogie also performs several smaller translation steps *prior* to the CFG-to-DAG phase. These include transforming ASTs to corresponding CFGs, optimisations such as dead variable elimination, and desugaring procedure calls using their specifications (via explicit **assert**, **assume**, and **havoc** statements). Our approach applies analogously to these initial smaller phases, but our current implementation certifies only the pipeline of all phases from the (input to the) CFG-to-DAG phase onwards. Thus, our certificate relates Boogie's VC to the original source AST program so long as these prior translation steps are correct.

*CFG Decomposition.* When tackling the certification of *each* phase, we further break down validation of a procedure's CFG in the source program of the phase into sub-problems for each block in the CFG. We prove two results for each block in the source CFG:

1. *Local block lemmas:* We prove an independent lemma for each source CFG block in isolation, relating the executions through the block with the corresponding block in the target program (or the VC generated for that block, in the case of the VC phase). In particular, this lemma implies that if the target block has no failing executions (or the VC generated for that block holds, for the VC phase), neither does the source block for corresponding input states.
2. *Global block theorems:* We show analogous per-block results concerning all executions *from this block onwards* extending to the end of the procedure in question; we build these compositionally by reverse-topological traversal of either the source or target CFGs, as appropriate. The global block theorem for the entry block establishes correctness of the phase.

This decomposition separates command-level reasoning (local block lemmas) from CFG-level reasoning (global block theorems). It enables concise lemmas and proofs in Isabelle and makes each comprehensible to a human.

## 3 A Formal Semantics for Boogie

Our certificates prove that the validity of a VC generated by Boogie formally implies correctness of the Boogie CFG-to-DAG source program. This proof relies crucially on a formal semantics for Boogie itself. Our first contribution is the first such formal semantics for a significant subset of Boogie, mechanised in Isabelle. Our semantics uses the Boogie reference manual [29], the presentation of its type system [33], and the Boogie implementation for reference; none of those provide a formal account of the language. For space reasons, we explain only the key concepts of our detailed formalisation here; more details are provided in App. A of the TR [37] and the full Isabelle mechanisation is available as part of our accompanying artifact [36].

### 3.1 The Boogie Language

Boogie programs consist of a set of top-level declarations of global variables and constants (the *global data*), axioms, uninterpreted (polymorphic) functions, type constructors, and procedures. A procedure declaration includes parameter, local-variable, and result-variable declarations (the *local data*), a pre- and post-condition, and a procedure body given as a CFG.[1] CFGs are formalised as usual in terms of basic blocks (containing a possibly-empty list of *basic commands*), and edges; semantically, execution after a basic block continues via any of its successors non-deterministically.

$$\begin{aligned} e ::=\ & x \mid \mathbf{false} \mid \mathbf{true} \mid i \mid e_1\ bop\ e_2 \mid uop(e) \mid f[\vec{\tau}](\vec{e}) \mid \mathbf{old}(e) \mid \\ & \forall x : \tau.\ e \mid \exists x : \tau.\ e \mid \forall_{ty}\, t.\ e \mid \exists_{ty}\, \tau.\ e \\ \tau ::=\ & Int \mid Bool \mid C(\vec{\tau}) \mid t \quad c ::= \ \mathbf{assume}\ e \mid \mathbf{assert}\ e \mid x := e \mid \mathbf{havoc}\ x \end{aligned}$$

**Fig. 2.** The syntax of our formalised Boogie subset, where $\tau$, $e$, and $c$, denote the types, expressions, and basic commands respectively; control-flow is handled via CFGs over the basic commands. *bop* and *uop* denote binary and unary operations, respectively.

The types, expressions, and basic commands in our Boogie subset are shown in Fig. 2. We support the primitive types *Int* and *Bool*; types obtained via declared type constructors are *uninterpreted types*; the sets of values such types denote are constrained only via Boogie axioms and **assume** commands. Moreover, types can contain type variables (for instance, to specify polymorphic functions).

Boogie expression syntax is largely standard (e.g. including typical arithmetic and boolean operations). Old-expressions **old**($e$) evaluate the expression $e$ w.r.t. the current local data and the global data as it *was* in the pre-state of the

[1] Source-level procedure specifications also include *modifies clauses*, declaring a set of global variables the procedure may modify. As we tackle Boogie programs after procedure calls have been desugared, there are no modifies clauses in our formalisation.

procedure execution. Boogie expressions also include universal and existential *value* quantification (written $\forall x : \tau.\ e$ and $\exists x : \tau.\ e$), as well as universal and existential *type* quantification (written $\forall_{ty}\, t.\ e$ and $\exists_{ty}\, t.\ e$). In the latter, $t$ is bound in $e$ and quantifies over *closed* Boogie types (i.e. types that do not contain any type variables).

Basic commands form the single-steps of traces through a Boogie CFG; sequential composition is implicit in the list of basic commands in a CFG basic block and further control flow (including loops) is prescribed by CFG edges. Boogie's basic commands are assumes, asserts, assignments, and havocs; **havoc** $x$ non-deterministically assigns a value matching the type of variable $x$ to $x$.

The main Boogie features *not* supported by our subset are maps and other primitive types such as bitvectors. Boogie maps are polymorphic and impredicative, i.e. one can define maps that contain themselves in their domain. Giving a semantic model for such maps in a proof assistant such as Isabelle or Coq is non-trivial; we aim to tackle this issue in the future. Modelling bitvectors will be simpler, although maintaining full automation may require some additional work.

## 3.2 Operational Semantics

*Values and State Model.* Our formalisation embeds integer and boolean values shallowly as their Isabelle counterparts; an Isabelle carrier type for all *abstract values* (those of uninterpreted types) is a parameter of our formalisation. Each uninterpreted type is (indirectly) associated with a *non-empty* subset of abstract values via a *type interpretation* map $\mathcal{T}$ from abstract values to (single) types; particular interpretations of uninterpreted types can be obtained via different choices of type interpretation $\mathcal{T}$.

One can understand Boogie programs in terms of the sets of possible *traces* through each procedure body. Traces are (as usual) composed of sequences of steps according to the semantics of basic commands and paths through the CFG; these can be finite or infinite (representing a non-terminating execution). A trace may halt in three cases: (1) an exit block of the procedure is reached in a state satisfying the procedure's postcondition (a *complete* trace),[2] (2) an **assert** $A$ command is reached in a state not satisfying assertion $A$ (a *failing* trace), or (3) an **assume** $A$ command is reached in a state not satisfying $A$ (a trace which *goes to magic* and stops). Our formalisation correspondingly includes three kinds of Boogie program states: a distinguished *failure state* $\mathsf{F}$, a distinguished *magic state* $\mathsf{M}$, and *normal states* $\mathsf{N}((os, gs, ls))$. A normal state is a triple of partial mappings from variables to values for the old global state (for the evaluation of old-expressions), the (current) global state, and the local state, respectively.

*Expression Evaluation.* An expression $e$ evaluates to value $v$ if the (big-step) judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle e, \mathsf{N}(ns)\rangle \Downarrow v$ holds in the context $(\mathcal{T}, \Lambda, \Gamma, \Omega)$. Here, $\mathcal{T}$

[2] The case of the postcondition *not* holding is subsumed under point (2), since Boogie checks postconditions by generating extra **assert** statements.

```
assume i != 0
j := 0
while(i != 0)
inv j >= 0 ∧ (i = 0 ⇒ j > 0)
{
  if(i < 5) {
    j := j+1
  }
  i := i-1
}
assert j > 0
```

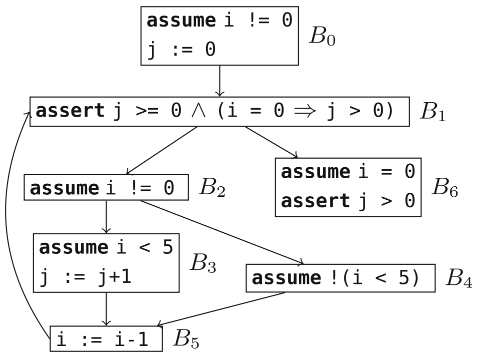


**Fig. 3.** Running example in source code and CFG representation, respectively.

is a *type interpretation* (as above), $\Lambda$ is a *variable context*: a pair $(G, L)$ of type declarations for the global $(G)$ and local $(L)$ data. $\Gamma$ is a *function interpretation*, which maps each function name to a semantic function mapping a list of types and a list of values to a return value. The type substitution $\Omega$ maps type variables to types.

The rules defining this judgement can be found in App. A.2 of the TR [37]. For example, the following rule expresses when a universal type quantification evaluates to **true** ($t$ is bound to the quantified type and may occur in $e$):

$$\frac{\forall \tau.\ \mathit{closed}(\tau) \Longrightarrow \mathcal{T}, \Lambda, \Gamma, \Omega(t \mapsto \tau) \vdash \langle e, \mathit{ns}\rangle \Downarrow \texttt{true}}{\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle \forall_{ty}\, t.\ e, \mathit{ns}\rangle \Downarrow \texttt{true}}$$

The premise requires one to show that the expression $e$ reduces to **true** for every possible type $\tau$ that is closed. In general, expression evaluation is possible only for well-typed expressions; we also formalise Boogie's type system and (for the first time) prove its type safety for expressions in Isabelle.

*Command and CFG Reduction.* The (big-step) judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle c, s\rangle \rightarrow s'$ defines when a command $c$ reduces in state $s$ to state $s'$; the rules are in App. A.3 of the TR [37]. This reduction is lifted to lists of commands $cs$ to model the semantics of a single trace through a CFG block (the judgement $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs, s\rangle\ [\rightarrow]\ s'$). The operational semantics of CFGs is modelled by the (small-step) judgement $\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash \delta \rightarrow_{\mathsf{CFG}} \delta'$, expressing that the CFG configuration $\delta$ reduces to configuration $\delta'$ in the CFG $G$. A CFG configuration is either *active* or *final*. An active configuration is given by a tuple $(\mathsf{inl}(b_n), s)$, where $b_n$ is the block identifier indicating the current position of the execution and $s$ is the current state. A final configuration consists of a tuple $(\mathsf{inr}(()), s)$ for state $s$ (and unit value $()$) and is reached at the end of a block that has either no successors, or is in a magic or failure state.

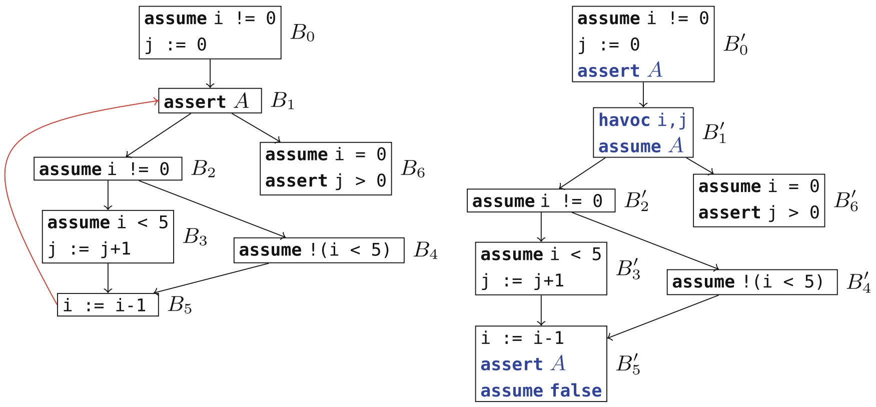


**Fig. 4.** The CFG-to-DAG phase applied to the running example (source is left, target is right). The back-edge (the red edge from $B_5$ to $B_1$ in the left CFG) is eliminated. The blue commands are new. $A$ is given by `j >= 0` $\wedge$ `(i = 0` $\Rightarrow$ `j > 0)`.

## 3.3 Correctness

A procedure is *correct* if it has *no failing traces*. This is a *partial correctness* semantics; a procedure body whose traces never leave a loop is trivially correct provided that no intermediate **assert** commands fail. Procedure correctness relies on CFG correctness. A CFG $G$ is correct w.r.t. a postcondition $Q$ and a context $(\mathcal{T}, \Lambda, \Gamma, \Omega)$ in an initial normal state $\mathsf{N}(ns)$ if the following holds for all configurations $(r, s')$:

$$\mathcal{T}, \Lambda, \Gamma, \Omega, G \vdash (\mathsf{inl}(\mathsf{entry}(G)), \mathsf{N}(ns)) \rightarrow^{*}_{\mathsf{CFG}} (r, s') \Longrightarrow [s' \neq \mathsf{F} \wedge \\ (r = \mathsf{inr}(())) \Longrightarrow (\forall ns'.\ s' = \mathsf{N}(ns') \Longrightarrow \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle Q, \mathsf{N}(ns')\rangle \Downarrow \mathbf{true}))]$$

where $\mathsf{entry}(G)$ is the entry block of $G$ and $\rightarrow^{*}_{\mathsf{CFG}}$ is the reflexive-transitive closure of the CFG reduction. The postcondition is needed only if a final configuration is reached in a normal state, while failing states must be unreachable. Whenever we omit $Q$, we implicitly mean the postcondition to be simply **true**. In our tool, we consider only empty initial mappings $\Omega$, since we do not support procedure type parameters (lifting our work to this feature will be straightforward).

For a procedure $p$ to be correct w.r.t. a context, its body CFG must be correct w.r.t. the same context and $p$'s postcondition, *for all* initial normal states $\mathsf{N}(ns)$ that satisfy $p$'s precondition and which respect the context. For $ns$ to *respect* a context, it must be well-typed and must satisfy the axioms when restricted to its constants. We say that $p$ is *correct*, if it is correct w.r.t. *all well-formed contexts*, which must have a well-typed function interpretation and a type interpretation that inhabits every uninterpreted closed type (and only those).

*Running Example.* We will use the simple CFG of Fig. 3 as a running example, intended as body of a procedure with trivial (**true**) pre- and post-conditions.

The code includes a simple loop with a declared loop invariant, which functions as a classical Floyd/Hoare-style inductive invariant, and for the moment can be considered as an implicit **`assert`** statement at the loop head. The CFG has infinite traces: those which start from any state in which `i` is negative. Traces starting from a state in which `i` is zero go to magic; they do not reach the loop. The program is correct (has no failing traces): all other initial states will result in traces that satisfy the loop invariant and the final **`assert`** statement. If we removed the initial **`assume`** statement, however, there *would* be failing traces: the loop invariant check would fail if `i` were initially zero.

## 4 The CFG-to-DAG Phase

In this section, we present the validation for the CFG-to-DAG phase in the Boogie verifier. This phase is challenging as it changes the CFG structure, inserts additional non-deterministic assignments and **`assume`** statements, and must do so correctly for arbitrary (reducible) nested loop structures, which can include unstructured control flow (e.g. jumps out of loops).

### 4.1 CFG-to-DAG Phase Overview

The CFG-to-DAG phase applies to every *loop head* block identified by Boogie's implementation and any *back-edges* from a block reachable from the loop head block back to the loop head (following standard definitions for reducible CFGs [21]). Figure 4 illustrates the phase's effect on our running example. Block $B_1$ is the (only) loop head here, and the edge from $B_5$ to it the only back-edge (completing looping paths via $B_2$ and $B_3$ or $B_2$ and $B_4$). An **`assert`** $A$ statement starting a loop head (like $B_1$) is interpreted as declaring $A$ to be the loop invariant.[3] The CFG-to-DAG phase performs the following steps:

1. Accumulate a set $X_H$ of all (local and global) variables *assigned-to* on *any looping path* from the loop head back to itself. In our example, $X_H$ is $\{i, j\}$.
2. Move the **`assert`** $A$ statement declaring a loop invariant (if any) from the loop head to the end of *each preceding* block (in our example: $B_0$ and $B_5$).
3. Insert **`havoc`** statements at the start of the loop head block per variable in $X_H$, followed by a single **`assume`** $A$ statement (preceding any further statements).
4. For each block with a back-edge to the loop head, delete the back-edge; if this leaves the block with no successors, append **`assume false`** to its commands.[4]

The havoc-then-assume sequence introduced in step 3 can be understood as generating traces for *arbitrary values of* $X_H$ satisfying the loop invariant $A$,

[3] In general, multiple asserts at the beginning of a loop head may form the invariant.

[4] Omitting **`assume false`** if there are no successors would be incomplete, since otherwise the postcondition would have to be satisfied.

effectively over-approximating the set of states reachable at the loop head in the original program. In particular, the remnants of any originally looping path (e.g. $B'_1$, $B'_2$, $B'_3$, $B'_5$) enforce that any non-failing trace starting from any such state must (due to the **`assert`** added to block $B'_5$ in step 2) result in a state which re-establishes the loop invariant. Such paths exist only to enforce this inductive step (analogously to the premise of a Hoare logic while rule); so long as the **`assert`** succeeds, we can discard these traces via step 4.

While we illustrate this step on a simple CFG, in general a loop head may have multiple back-edges, looping structures may nest, and edges may exit multiple loops. For the above translation to be correct, the CFG must be reducible and loop heads and corresponding back-edges identified accurately, which is complex in general. Importantly (but perhaps surprisingly), our work makes this phase of Boogie certifying *without* explicitly verifying (or even defining) these notions.

## 4.2 CFG-to-DAG Certification: Local Block Lemmas

We define first our local block lemmas for this phase. Recall that these prove that if executing the statements of a target block yields no failing executions, the same holds for the corresponding source block; this result is trivial for source blocks other than loop heads and their immediate predecessors, since these are unchanged in this phase. To enable eventual composition of our block lemmas, we need to also reflect the role of the **`assume`** and **`assert`** statements employed in this phase. The formal statement of our local block lemmas is as follows[5]:

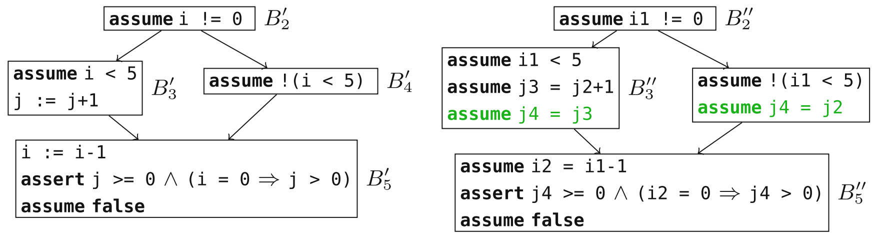
**Theorem 1 (CFG-to-DAG Local Block Lemma).** *Let $B$ be a source block with commands $cs_S$, whose corresponding target block has commands $cs_T$. If $B$ is a loop head, let $X_H$ be as defined in CFG-to-DAG step 1 (and empty otherwise) and let $A_{pre}$ be its loop invariant (or* **`true`** *otherwise). If $B$ is a* predecessor *of a loop head, let $A_{post}$ be the loop invariant of its successor (and* **`true`** *otherwise). Then,* if*:*

1. $\mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs_S, \mathsf{N}(ns_1)\rangle \; [\rightarrow] \; s'_1$
2. $\forall s'_2.\; \mathcal{T}, \Lambda, \Gamma, \Omega \vdash \langle cs_T, \mathsf{N}(ns_2)\rangle \; [\rightarrow] \; s'_2 \Longrightarrow s'_2 \neq \mathsf{F}$
3. *$A_{pre}$ is satisfied in $ns_1$, and $ns_2$ differs from $ns_1$ only on variables in $X_H$ and variables not defined in $\Lambda$*

then*: $s'_1 \neq \mathsf{F}$ and if $s'_1$ is a normal state, then (1) $A_{post}$ is satisfied in $s'_1$, and (2) if no* ***`assume false`*** *was added at the end of $cs_T$, then there is a target execution in $cs_T$ from $\mathsf{N}(ns_2)$ that reaches a normal state that differs from $s'_1$ only on variables not defined in $\Lambda$.*

The gist of this lemma is to capture *locally* the ideas behind the four steps of the phase. For example, consequence (1) reflects that *after* the transformation, any blocks that *were previously* predecessors of a loop head ($B'_0$ and $B'_5$ in our running example) will have an **`assert`** statement checking for the corresponding invariant (and so if the target program has no failing traces, in each trace this invariant will be true at that point).

[5] We omit some details regarding well-typedness, handled fully in our formalisation.



```
assume i != 0

assume i < 5
j := j+1

assume !(i < 5)

i := i-1
assert j >= 0 ∧ (i = 0 ⇒ j > 0)
assume false
```

```
assume i1 != 0

assume i1 < 5
assume j3 = j2+1
assume j4 = j3

assume !(i1 < 5)
assume j4 = j2

assume i2 = i1-1
assert j4 >= 0 ∧ (i2 = 0 ⇒ j4 > 0)
assume false
```

**Fig. 5.** The passification phase applied to the branch in the running example with the result on the right. The final (green) commands in $B''_3$ and $B''_4$ are the synchronisation commands. At the uppermost blocks shown here, the current versions of `i` and `j` are `i1` and `j2`, respectively. The full CFGs are shown in App. B of the TR [37].

## 4.3 CFG-to-DAG Certification: Global Block Theorems

We lift our certification to *all* traces through the source and target CFGs; the statement of the corresponding global block theorems is similar to that of local block theorems lifted to CFG executions, and for space reasons we do not present it here, but it is included in our Isabelle formalisation. In particular, we prove for each block (working in reverse topological order through the target CFG blocks) that if executions starting in the target CFG block never fail, neither do any executions starting from the corresponding source CFG block, and looping paths modify at most the variables havoced according to step 3 of the phase.

The major challenge in these proofs is reasoning about looping paths in the source CFG, since these revisit blocks. To solve this challenge, we perform inductive arguments per loop head in terms of the number of steps remaining in the trace in question.[6] Our global block theorem for a block $B$ then carries as an assumption an induction hypothesis for each loop that contains $B$. Proving a global block theorem for the origin of a back-edge is taken care of by applying the corresponding induction hypothesis.

This proof strategy works only if we have obtained the induction hypothesis for the loop head *before* we use the global block theorem of the origin of a back-edge (otherwise we cannot discharge the block theorem's hypothesis). In other words, our proof implicitly shows the necessary requirement that loop heads (as identified by Boogie) dominate all back-edges reaching them *without us formalising any notion of domination, CFG reducibility, or any other advanced graph-theoretic concept.* This shows a major benefit of our validation approach over a once-and-for-all verification of Boogie itself: our proofs indirectly check that the identification of loop heads and back-edges guarantees the necessary *semantic properties* without being concerned with *how* Boogie's implementation computes this information.

[6] This may seem insufficient since traces can be infinite, but importantly a *failing* trace is always finite, and our theorems need only eliminate the chance of failing traces.

Our approach applies equally to nested loops and more-generally to reducible CFG structures; *all* corresponding induction hypotheses are carried through from the visited loop heads. The requirement that no more than the havoced variables $X_H$ are modified in the source program is easily handled by showing that variables modified in an inner loop are a subset of those in outer loops. As for all of our results, our global block lemmas are proven automatically in Isabelle per Boogie procedure, providing per-run certificates for this phase.

## 5 The Passification Phase

In this section, we describe the validation of the passification phase in the Boogie verifier. Unlike the previous phase, passification makes no changes to the CFG structure, but makes substantial changes to the program states (via SSA-like renamings), substantially increases non-determinism, and employs **`assume`** statements to re-tame the sets of possible traces.

### 5.1 Passification Phase Overview

The main goal of passification is to eliminate assignments such that a more efficient VC can be ultimately generated [6,18,30]. In the Boogie verifier, this is implemented as a single transformation phase that can be thought of as two independent steps. Firstly, the source CFG is transformed into *static single assignment* (SSA) form, introducing *versions* (fresh variables) for each original program variable such that each version is assigned at most once in any program trace. In a second step, variable assignments are *completely eliminated*: each assignment command $x := e$ is replaced by **`assume`** $x = e$. Havoc statements are simply removed; their effect is implicit in the fact that a new variable version is used (via the SSA step) *after* such a statement.

Figure 5 shows the effect of this phase on four blocks of our running example (the full figure of the target CFG is shown in App. B of the TR [37]). The commands inserted just before the join block (here, $B_5''$) introduce a consistent variable version (here, `j4`) for use in the join block. It is convenient to speak of target variables in terms of their source program counterparts: we say e.g. that `j` *has version 4* on entry to block $B_5'$.

Compared to traces through the source program, the space of variable values in a trace through the target program is initially much larger; each version may, on entry to the CFG, have an arbitrary value. For example, `j4` may have any value on entry to $B_2''$; traces in which its value does not correspond to the constraint of the **`assume`** statements in $B_3''$ or $B_4''$ will go to magic and not reach $B_5''$. Importantly, however, not *all* traces go to magic; enough are preserved to simulate the executions of the original program: each **`assume`** statement constrains the value of exactly one variable version, and the same version is never constrained more than once. Capturing this delicate argument formally is the main challenge in certifying this step.

As extra parts of the passification phase, the Boogie verifier performs constant propagation and desugars old-expressions (using variable versions appropriate to the entry point of the CFG). We omit their descriptions here for brevity, but our implementation certifies them.

## 5.2 Passification Certification: Local Block Lemmas

To validate the passification phase, it is sufficient to show that each source execution is simulated by a corresponding target execution, made precise by constructing a relation between the states in these executions. Such *forward simulation* arguments are standard for proving correctness of compilers for deterministic languages. However, the situation here is more complex due to the fact that the target CFG has a much wider space of traces: the values of each versioned variable in the target program are initially unconstrained, meaning traces exist for all of their combinations. On the other hand, many of these traces do not survive the **`assume`** statements encountered in the target program. Picking the correct *single* trace or state to simulate a particular source execution would require knowledge of all variable assignments that are *going* to happen, which is not possible due to non-determinism and would preclude the block-modular proof strategies that our validation approach employs.

Instead, we generalise this idea to relating each single source state $s$ with a *set* $T$ of corresponding target program states. We define variable relations $\mathcal{V}_R$ at each point in a trace, making explicit the mappings used in the SSA step between source program variables and their corresponding versions. For example, on entry to block $B'_2$ in the source version of our running example (correspondingly $B''_2$ in the target), the $\mathcal{V}_R$ relation relates `i` to `i1` and `j` to `j2`. All states $t \in T$ must precisely agree with $s$ w.r.t. $\mathcal{V}_R$ (e.g., $s(\texttt{i}) = t(\texttt{i1})$, $s(\texttt{j}) = t(\texttt{j2})$). On the other hand, our sets of states $T$ are defined to be completely unconstrained (besides typing) for *future* variable versions. For example, for every $t \in T$ at the same point in our example, there will be states in $T$ assigning each possible value (of the same type) to `i2` (and otherwise agreeing with $t$).

More precisely, for a set of variables $X$, we say that a set of states $T$ *constrains at most $X$ w.r.t. variable context $\Lambda$* if, for every $t \in T$, $z \notin X$, $z$ is in $\Lambda$, and value $v$ of $z$'s type, we have $t[z \mapsto v] \in T$. In other words, the set $T$ is closed under arbitrary changes to values of all variables in $\Lambda$ but *not* in $X$. We construct our sets $T$ such that they constrain at most *current and past versions* of program variables. It is this fact that enables us to handle subsequent **`assume`** statements in the target program and, in particular, to show that the set of possible traces in the target program never becomes empty while there are possible traces in the source program. For example, when relating the source command `j := j+1` in $B'_3$ with the target command **`assume`** `j3 = j2 + 1` in block $B''_3$, we use the fact that our set of states does not constrain `j3` to prove that, although many traces go to magic at this point, for a non-empty set of states $T' \subseteq T$ (those in which `j3` has the "right" value equal to `j2 + 1`), execution continues in the target.

We now make these notions more precise by showing the definition of our local block lemmas for the passification phase (See footnote 5).

**Theorem 2 (Passification Local Block Lemma).** *Let $B$ be a source block with commands $cs$, whose corresponding target block has commands $cs'$; let $\mathcal{V}_R$ and $\mathcal{V}'_R$ be the variable relations at the beginning and end of $B$, respectively. Let $X$ be a set of variable versions, and $\mathsf{N}(ns)$ be a normal state. Let $T$ be a non-empty set of normal states such that $\mathsf{N}(ns)$ agrees with $T$ according to $\mathcal{V}_R$, and $T$ constrains at most $X$ w.r.t. $\Lambda_2$. Furthermore, let $Y$ be the variable versions corresponding to the targets of assignment and havoc statements in $cs$. If both*

1. $A, \Lambda_1, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle \; [\rightarrow] \; s' \wedge s' \neq \mathsf{M}$
2. $X \cap Y = \emptyset$

*then there exists a non-empty set of normal states $T' \subseteq T$ s.t. $T'$ constrains at most $X \uplus Y$ w.r.t. $\Lambda_2$ and for each $t' \in T'$, there exists a state $t'^*$ s.t.*

1. $A, \Lambda_2, \Gamma, \Omega \vdash \langle cs_2, t'\rangle \; [\rightarrow] \; t'^* \wedge (s' = \mathsf{F} \Longrightarrow t'^* = \mathsf{F})$
2. *If $s'$ is a normal state, then $s'$ and $t'$ are related w.r.t. $\mathcal{V}'_R$ (and $t'^* = t'$).*

This lemma captures our generalised notion of forward simulation appropriately. The first conclusion expresses that the target does not get stuck and that failures are preserved, while the second shows that if the source execution neither fails nor stops then the resulting states are related. Note that premise 2 is essential in the proof to guarantee that the **`assume`** statements introduced by passification do not eliminate the chance to simulate source executions; the condition expresses that the variable versions newly constrained do not intersect with those previously constrained. To prove these lemmas over the commands in a single block, we are forced to check that the same version is not constrained twice.

## 5.3 Passification Certification: Global Block Theorems

As for all phases, we lift our local block lemmas to theorems certifying all executions *starting* from a particular block, and thus, ultimately, to entire CFGs. For the passification phase, most of the conceptual challenges are analogous to those of the local block lemmas; we similarly employ $\mathcal{V}_R$ relations between source variables and their corresponding target versions. To connect with our local block lemmas (and build up our global block theorems, which we do backwards through the CFG structure), we repeatedly require the key property that the set of variable versions constrained in our executions so far is disjoint from those which may be constrained by a subsequent **`assume`** statement (*cf.* premise 2 of our local block lemma above). Concretely tracking and checking disjointness of these concrete sets of variables is simple, but turns out to get expensive in Isabelle when the sets are large.

We circumvent this issue with our own *global versioning scheme* (as opposed to the versions used by Boogie, which are *independent* for different source variables): according to the CFG structure, we assign a *global* version number $\mathsf{ver}_G(x)$

to each variable $x$ in the target program such that, if $x$ is constrained in a target block $B'$ and $y$ is constrained in another target block $B''$ reachable from $B'$, then $\mathsf{ver}_{\mathcal{G}}(x) < \mathsf{ver}_{\mathcal{G}}(y)$. Such a consistent global versioning always exists in the target programs generated by Boogie because the only variables not constrained exactly once *in the program* are those used to synchronise executions (i.e. `j4` in Fig. 5), which always appear right before branches are merged. We can now encode our disjointness properties much more cheaply: we simply compare the *maximal* global version of all already-constrained variables with the *minimal* global version of those (potentially) to be constrained. Since we represent variables as integers in the mechanisation, we directly use our global version *as* the variable name for the target program; there is no need for an extra lookup table. Note that (readability aside) it makes no difference which variables names are used in intermediate CFGs; we ultimately care only about validating the original CFG.

## 6 The VC Phase

In this section, we present the validation of the VC phase in the Boogie verifier. This phase has two main aspects: (1) it encodes and desugars all aspects of the Boogie type system, employing additional uninterpreted functions and axioms to express its properties [33]; program expression elements such as Boogie functions are analogously desugared in terms of these additional uninterpreted functions, creating a non-trivial logical gap between expressions as represented in the VC and those from the input program. (2) It performs an efficient (block-by-block) calculation of a weakest precondition for the (acyclic, passified) CFG, resulting in a formula characterising its verification requirements, subject to background axioms and other hypotheses.

### 6.1 VC Structure

The generated VC has the following overall structure (represented as a shallow embedding in our certificates)[7]:

$$\forall \underbrace{\textit{VC quantifiers}}_{\substack{\text{type encoding parameters,}\\ \text{functions, variable values}}} . \ (\underbrace{\textit{VC assumptions}}_{\substack{\text{type encoding,}\\ \text{func./var./prog. axioms}}} \Longrightarrow \textit{CFG WP})$$

The VC quantifies over parameters required for the type encoding, as well as VC counterparts representing the variable values and functions in the Boogie program. The VC body is an implication, whose premise contains: (1) assumptions that axiomatise the type encoding parameters, (2) axioms expressing the typing of Boogie variables and functions, and (3) assumptions directly relating

[7] Note that top-level quantification over functions is implicit in the (first-order) SMT problem generated by Boogie; we quantify explicitly in our Isabelle representation.

to axioms explicitly declared in the Boogie program. The conclusion of the implication is an optimised version of the weakest (liberal) precondition (WP) of the CFG.[8]

## 6.2 Boogie's Logical Encoding of the Boogie Type System

We first briefly explain Boogie's logical encoding of its own type system. Values and types are represented at the VC level by two uninterpreted carrier sorts $V$ and $T$. An uninterpreted function $\mathit{typ}$ from $V$ to $T$ maps each value to the representation of its type. Boogie type constructors are each modelled with an (injective) uninterpreted function $C$ with return sort $T$ and taking arguments (per constructor parameter) of sort $T$. For example, a type constructor $\mathit{List}(t)$ is represented by a VC function from $T$ to $T$. Projection functions are also generated for each type constructor ($C_i^{\pi}$ for each type argument at position $i$), e.g. mapping the representation of a type $\mathit{List}(t)$ to the representation of type $t$.

This encoding is then used in the VC to recover Boogie typing constraints for the untyped VC terms. Recovering the constraints is not always straightforward due to optimisations performed by Boogie. For example, the VC translation of the Boogie expression $\forall_{ty}\, t.\ \forall x : \mathit{List}(t).\ e$ no longer quantifies over types; all original occurrences of $t$ in $e$ having been translated to $\mathit{List}_1^{\pi}(\mathit{typ}(x))$. This optimisation reflects that this particular type quantification is redundant, since $t$ can be recovered from the type of $x$.[9]

## 6.3 Working from VC Validity

Our certificates assume that the generated VC is valid (certifying the validity-checking of the VC by an SMT solver is an orthogonal concern). However, connecting VC validity back to block-level properties about the specific program requires a number of technical steps. We need to construct Isabelle-level semantic values to *instantiate* the top-level quantifiers in the VC such that the corresponding VC assumptions (left-hand side of the VC) can be proved and, thus, validity of the corresponding WP can be deduced. Moreover, we must ensure that our instantiation yields a WP whose validity implies correctness of the Boogie program. For example, a top-level VC quantifier modelling a Boogie function $f$ must be instantiated with a mathematical function that behaves in the same way as $f$ for arguments of the correct type.

We instantiate the carrier sort $V$ for values in the VC with the corresponding type denoting Boogie values in our formalisation; the carrier sort $T$ for *types* is instantiated to be all Boogie types that do not contain free variables (i.e. closed types). Constructing explicit models for the quantified functions used to

[8] One difference in our version of the Boogie verifier is that we switched off the generation of extra variables introduced to report error traces [32]; these are redundant for programs that do not fail and further complicate the VC structure.

[9] Note that in the VC the quantification over $x$ ranges over all values of sort $V$. An implication is used to consider only those $x$ for which $\mathit{typ}(x) = \mathit{List}(\mathit{List}_1^{\pi}(\mathit{typ}(x)))$.

model Boogie's type system (satisfying, e.g., suitable inverse properties for the projection functions) is straightforward. For the VC-level variable values, we can directly instantiate the corresponding values in the initial Boogie program state.

VC-level functions representing those declared in the Boogie program are instantiated as (total) functions which, *for input values of appropriate type* (the arguments and output are untyped values of sort $V$), are defined simply to return the same values as the corresponding function in our model. However, perhaps surprisingly, Boogie's VC embedding of functions logically requires functions to return values of the specified return type even if the input values do not have the types specified by the function. In such cases, we define the instantiated function to return some value of the specified type, which is possible since in well-formed contexts every closed type has at least one value in our model.

After our instantiation, we need to prove the hypotheses of the VC's implication; in particular that all axioms (both those generated by the type system encoding and those coming from the program itself) are satisfied. The former are standard and simple to prove (given the work above), while the latter largely follow from the assumption that each declared axiom must be satisfied in the initial state restricted to the constants. The only remaining challenge is to relate VC expressions with the evaluation of corresponding Boogie expressions; an issue which also arises (and is explained) below, where we show how to connect validity of the instantiated WP to the program.

## 6.4 Certifying the VC Phase

Boogie's weakest precondition calculation is made size-efficient by the usage of explicit named constants for the weakest preconditions $wp(B, \texttt{true})$ for each block $B$, which is defined in terms of the named constants for its successor blocks. For example, in Fig. 5, $wp(B_2'', \texttt{true})$ is given by $i_1^{vc} \neq 0 \implies wp(B_3'', \texttt{true}) \wedge wp(B_4'', \texttt{true})$. Here $i_1^{vc}$ is the value that we instantiated for the variable `i1`.

We exploit this modular construction of the generated weakest precondition for the local and global block theorems. We prove for each block $B$ with commands $cs$ the following local block lemma:

**Theorem 3 (VC Phase Local Block Lemma).**
*If $A, \Lambda, \Gamma, \Omega \vdash \langle cs, \mathsf{N}(ns)\rangle \; [\rightarrow] \; s'$ and $wp(B, \texttt{true})$ holds, then $s' \neq \mathsf{F}$ and if $s'$ is a normal state, then $\forall B_{suc} \in successors(B).\ wp(B_{suc}, \texttt{true})$.*

Once one has proved this lemma for all blocks in the CFG, combining them to obtain the corresponding global block theorems (via our usual reverse walk of the CFG) is straightforward. The main challenge is in decomposing the proof for the local block lemma itself for a block $B$, for which we outline our approach next.

By this phase, the first command in $B$ must be either an **`assume`** $e$ or an **`assert`** $e$ command. In the former case, we rewrite $wp(B, \texttt{true})$ into the form $e^{vc} \implies H$, where $e^{vc}$ is the VC counterpart of $e$ and where $H$ corresponds to the weakest precondition of the remaining commands. This rewriting may involve undoing certain optimisations Boogie's implementation performed on the formula structure. Next, we need to prove that $e$ evaluates to $e^{vc}$ (see below). Hence, if $e$ evaluates to **true** (the execution does not go to magic) then $H$ must be true, and we can continue inductively. The argument for **assert** $e$ is similar but where we rewrite the VC to $e^{vc} \wedge H$ (i.e. $e^{vc}$ and $H$ must both hold); if $e$ evaluates to $e^{vc}$, we know that the execution does not fail.

Proving that $e$ evaluates to $e^{vc}$ arises in both cases and also in our previous discharging of VC hypotheses. Note that, in contrast to $e$, $e^{vc}$ is not a Boogie expression, but a shallowly embedded formula that includes the instantiations of quantified variables we constructed above. Showing this property works largely on syntax-driven rules that relate a Boogie expression with its VC counterpart, except for extra work due to mismatching function signatures and optimisations that Boogie made either to the formula structure or via the type system encoding (*cf.* Sect. 6.2). We handle some of these cases by showing that we can rewrite the formula back into the unoptimised standard form we require for our syntax-driven rules and in other cases we directly work with the optimised form. Both cases are automated using Isabelle tactics.

This concludes our discussion of the certification of Boogie's three key phases. Combining the three certificates yields an end-to-end proof that the validity of the generated verification conditions implies the correctness of the input program, that is, that the given verification run is sound.

## 7 Implementation and Evaluation

In this section, we evaluate our certifying version of the Boogie verifier [36], which produces Isabelle certificates proving the correctness of Boogie's pipeline for programs it verifies.

We have implemented our validation tool as a new C# module compiled with Boogie. We instrumented Boogie's codebase to call out to our module, which allows us to obtain information that we can use to validate the key phases, and extended parts of the codebase to extract information more easily. Moreover, we disabled counter-example related VC features and the generation of VC axioms for any built-in types and operators that we do not support. We added or changed fewer than 250 non-empty, uncommented lines of code across 11 files in the existing Boogie implementation.

Given an input file verified by Boogie, our work produces an Isabelle certificate per procedure $p$ that certifies the correctness of the corresponding CFG-to-DAG source CFG as represented internally in Boogie. The generation and checking of the certificate is fully automatic, without any user input. We use a combination of custom and built-in Isabelle tactics. In addition to the three key phases we describe in detail, our implementation also handles several smaller transformations made by Boogie, such as constant propagation. Our tool currently supports the default options of Boogie (only) and does not support advanced source-level *attributes* (for instance, to selectively force procedures to be inlined).

**Table 1.** Selection of algorithmic examples with the lines of code (LOC), the number of procedures (#P), the time it takes for Isabelle to check the certficate in seconds (the average of 5 runs on a Lenovo T480 with 32 GB, i7-8550U 1.8 GhZ, Ubuntu 18.04 on the Windows Subsystem for Linux), and the certificate size expressed as the number of non-empty lines of Isabelle.

| Name | LOC | #P | Time [s] | Size |
|---|---|---|---|---|
| TuringFactorial | 29 | 1 | 19.4 | 1986 |
| Find | 27 | 2 | 27.3 | 2100 |
| DivMod | 69 | 2 | 28.4 | 4753 |
| Summax [27] | 23 | 1 | 19.1 | 1953 |
| MaxOfArray [12] | 22 | 1 | 19.9 | 1944 |
| SumOfArray [12] | 22 | 1 | 18.7 | 1534 |
| Plateau [12] | 50 | 1 | 22.9 | 2019 |
| WelfareCrook [12] | 52 | 1 | 39.4 | 2528 |
| ArrayPartitioning [12] | 57 | 2 | 27.6 | 3514 |
| DutchFlag [12] | 76 | 2 | 52.8 | 3994 |

We evaluated our work in two ways. Firstly, to evaluate the applicability of our certificate generation, we automatically collected all input files with at least one procedure from Boogie's test suite [1] which verify successfully and which either use no unsupported features or are easily desugared (by hand) into versions without them. This includes programs with procedure calls since Boogie simply desugars these in an early stage. For programs employing attributes, we checked whether the program still verifies *without* attributes, and if so we also kept these. In total, this yields 100 programs from Boogie's test suite. Secondly, we collected a corpus of ten Boogie programs which verify interesting algorithms with non-trivial specifications: three from Boogie's test suite and seven from the literature [12,27]. Where needed we manually desugared usages of Boogie maps (which we do not yet support) using type declarations, functions, and axioms.

Of the 100 programs from Boogie's test suite, we successfully generate certificates in 96 cases. The remaining 4 cases involve special cases that we do not handle yet. For 2 of them, extending our work is straightforward: one special case includes a naming clash and the other case can be amended by using a more specific version of a helper lemma. The remaining two fail because of our incomplete handling of function calls in the VC phase when combined with coercions between VC integers or booleans and their Boogie counterparts. Handling this is more challenging but is not a fundamental issue.

For the corpus of 10 examples, Table 1 shows the generated certificate size and the time for Isabelle to check their validity.[10] The ratio of certificate size to code size ranges from 41 to 89; this rather large ratio emphasises the substantial work in formally validating the substantial work which Boogie's implementation

[10] The time to generate the certificate is not included, but is negligible here.

performs. Optimisations to further reduce the ratio are possible. The validation of certificates takes usually under one second per line of code. While these times are not short, they are acceptable since certificate generation needs to run only for (verified) release versions of the program in question.

## 8 Related Work

Several works explore the validation of program verifiers. Garchery et al. [20] validate VC rewritings in the Why3 VC generator [16]. Unlike our work, they do not connect VCs with programs and do not handle the erasure of polymorphic types. Strub et al. [39] validate part of a previous version of the F* verifier [40] by generating a certificate for the F* type checker itself, which type checks programs by generating VCs. Like us, they assume the validity of the generated VC itself, but they do not consider program-to-program transformations such as ours. Another approach is taken by Aguirre [2] who shows how one can map proofs of the VC back to correctness of an F* program. They prove a once-and-for-all result, but the approach could be lifted to a validation approach using the proof-producing capability of SMT solvers [7]. Lifting the approach would require extending the work to handle classical instead of constructive VC proofs.

There is some work on proving VC generator implementations correct once and for all, although none of the proven tools are used in practice. Homeier and Martin [23] prove a VC generator correct in HOL for an executable language and a simpler VC phase than Boogie's. Herms et al. [22] prove a VC generator inspired by Why3 correct in Coq. However, some more-challenging aspects of Why3's VC transformation and polymorphic type system are not handled. Vogels et al. [44] prove a toolchain for a Boogie-like language correct in Coq, including passification and VC phases. However, the language is quite limited: without unstructured control flow, loops (i.e. no need for a CFG-to-DAG phase), functions, or polymorphism (i.e. no type encoding). Verifiers other than VC generators, include the verified Verasco static analyzer [25], which supports a realistic subset of C, but whose performance is not yet on par with unverified, industrial analyzers.

Validation has also been explored in other settings. Alkassar et al. [3] adjust graph algorithms to produce witnesses that can be then used by verified validators to check whether the result is correct. In the context of compiler correctness, many validation techniques express a per-run validator in Coq, prove it correct once-and-for-all [8,41,43], and then extract executable code (the extraction must be trusted). In the verified CompCert compiler [34], such validators have been used in combination with the once-and-for-all approach. Validators are used for phases that can be more easily validated than proved correct once and for all. One such example related to our certification of the passification phase is the validation of the SSA phase [8], dealing also with versioned variables in the target (but not with **assume** statements that prune executions). In contrast to our work, they require an explicit notion of CFG domination and they do not use a global versioning scheme to efficiently check that two parts of the CFG constrain disjoint versions. Our versioning idea is similar to a technique used for the validation of a dominator relation in a CFG [9], which assigns intervals to basic blocks (as opposed to assigning versions to variables) to efficiently determine whether a block dominates another one. The validation of the Cogent compiler [38] follows a similar approach to ours in that it generates proofs in Isabelle.

## 9 Conclusion

We have presented a novel verifier validation approach, and applied it successfully to three key phases of the Boogie verifier, providing formal underpinnings for both the language and its verifier for the first time. Our work demonstrates that it is feasible to provide strong formal guarantees regarding the verification results of practical VC generators written in modern mainstream languages.

In the future, we plan to extend our supported subset of Boogie, e.g. to include procedure calls and bitvectors. Supporting Boogie's potentially-impredicative maps is the main open challenge: maps can take other maps as input, potentially including themselves. The challenge with this feature is to still be able to express a type in Isabelle capturing all Boogie values despite the potentially-cyclic nature of map types. In practice, however, this may not be required in full generality: we have observed that Boogie front-ends rarely use maps that contain maps of the same type as input. Therefore, we plan to extend our technique to support a suitably-expressive restricted form of Boogie maps.

**Acknowledgements.** We thank Alain Delaët–Tixeuil for his earlier work on this topic, Thibault Dardinier for improving our artifact, Martin Clochard for helpful discussions and the anonymous reviewers for their valuable comments. This work was partially funded by the Swiss National Science Foundation (SNSF) under Grant No. 197065.

## References

1. Boogie verifier repository. https://github.com/boogie-org/boogie
2. Aguirre, A.: Towards a provably correct encoding from F* to SMT. Technical report, INRIA (2016)
3. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A framework for the verification of certifying computations. JAR **52**(3), 241–273 (2014)
4. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. In: OOPSLA (2019)
5. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. CACM **54**(6), 81–91 (2011)
6. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE (2005)
7. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: All about Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55, pp. 23–44. College Publications (2015)
8. Barthe, G., Demange, D., Pichardie, D.: Formal verification of an SSA-based middle-end for compcert. TOPLAS **36**(1), 1–35 (2014)

9. Blazy, S., Demange, D., Pichardie, D.: Validating dominator trees for a fast, verified dominance test. In: ITP (2015)
10. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: iFM (2007)
11. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: ITP (2010)
12. Chen, Y., Furia, C.A.: Triggerless happy - intermediate verification with a first-order prover. In: iFM (2017)
13. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: TPHOLs (2009)
14. Coq Development Team, T.: The Coq Reference Manual, version 8.10, available electronically at (2019). http://coq.inria.fr/documentation
15. Ekici, B., et al.: SMTCoq: a plug-in for integrating SMT solvers into Coq. In: CAV (2017)
16. Filliâtre, J.C., Paskevich, A.: Why3 – where programs meet provers. In: ESOP (2013)
17. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: CAV (2007)
18. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: POPL (2001)
19. Fleury, M., Schurr, H.: Reconstructing veriT proofs in Isabelle/HOL. In: PxTP (2019)
20. Garchery, Q., Keller, C., Marché, C., Paskevich, A.: Des transformations logiques passent leur certificat. In: JFLA (2020)
21. Hecht, M.S., Ullman, J.D.: Flow graph reducibility. SIAM J. Comput. **1**(2), 188–202 (1972)
22. Herms, P., Marché, C., Monate, B.: A certified multi-prover verification condition generator. In: VSTTE (2012)
23. Homeier, P.V., Martin, D.F.: A mechanically verified verification condition generator. Comput. J. **38**(2), 131–141 (1995)
24. Isabelle Development Team, T.: The Isabelle Documentation, version June 2019, available electronically at (2019). https://isabelle.in.tum.de/documentation.html
25. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: POPL (2015)
26. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015)
27. Klebanov, V., et al.: The 1st verified software competition: Experience report. In: FM (2011)
28. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV (2012)
29. Leino, K.R.M.: This is Boogie 2 (June 2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
30. Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. **93**(6), 281–288 (2005)
31. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: LPAR (2010)
32. Leino, K.R.M., Millstein, T.D., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Sci. Comput. Program. **55**(1–3), 209–226 (2005)
33. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: design and logical encoding. In: TACAS (2010)

34. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL (2006)
35. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: VMCAI (2016)
36. Parthasarathy, G., Müller, P., Summers, A.J.: Formally validating a practical verification condition generator - artifact (2021). https://doi.org/10.5281/zenodo.4726554
37. Parthasarathy, G., Müller, P., Summers, A.J.: Formally validating a practical verification condition generator (extended version) (2021). arXiv:2105.14381
38. Rizkallah, C., et al.: A framework for the automatic formal verification of refinement from Cogent to C. In: ITP (2016)
39. Strub, P.Y., Swamy, N., Fournet, C., Chen, J.: Self-certification: Bootstrapping certified typecheckers in F* with Coq. In: POPL (2012)
40. Swamy, N., et al.: Dependent types and multi-monadic effects in F*. In: POPL (2016)
41. Tristan, J.B., Leroy, X.: Formal verification of translation validators: a case study on instruction scheduling optimizations. In: POPL (2008)
42. Tristan, J.B., Leroy, X.: Verified validation of lazy code motion. In: PLDI (2009)
43. Tristan, J.B., Leroy, X.: A simple, verified validator for software pipelining. In: POPL (2010)
44. Vogels, F., Jacobs, B., Piessens, F.: A machine-checked soundness proof for an efficient verification condition generator. In: SAC (2010)

# Automatic Generation and Validation of Instruction Encoders and Decoders

Xiangzhe Xu, Jinhua Wu, Yuting Wang(✉),
Zhenguo Yin, and Pengfei Li

Shanghai Jiao Tong University, Shanghai 200240, China
yuting.wang@sjtu.edu.cn

**Abstract.** Verification of instruction encoders and decoders is essential for formalizing manipulation of machine code. The existing approaches cannot guarantee the critical *consistency* property, i.e., that an encoder and its corresponding decoder are mutual inverses of each other. We observe that consistent encoder-decoder pairs can be automatically derived from bijections inherently embedded in instruction formats. Based on this observation, we develop a framework for writing specifications that capture these bijections, for automatically generating encoders and decoders from these specifications, and for formally validating the consistency and soundness of the generated encoders and decoders by synthesizing proofs in Coq and discharging verification conditions using SMT solvers. We apply this framework to a subset of X86-32 instructions to illustrate its effectiveness in these regards. We also demonstrate that the generated encoders and decoders have reasonable performance.

**Keywords:** Formalized instruction formats · Verified parsing · Program synthesis · Proof synthesis · Translation validation

## 1 Introduction

Software that manipulates machine code such as compilers, OS kernels and binary analysis tools, relies on *instruction encoders and decoders* for extracting structural information of instructions from machine code and for translating such information back into binary forms. Because of the sheer amount of instructions provided by any instruction set architecture (ISA) and the complexity of instruction formats, it is extremely tedious and error-prone to implement instruction encoders and decoders by hand. Therefore, the literature contains abundant work on automatic generation of instruction encoders and decoders, often from specifications written in a formal language capable of concisely and accurately characterizing instruction formats on various ISAs [7,12,15].

Unfortunately, the above approaches generate little formal guarantee, therefore not suitable for rigorous analysis or verification of machine code. In those settings, instruction encoders and decoders are expected to be *consistent*, i.e., any encoder and its corresponding decoder are inverses of each other, and *sound*, i.e., they meet formal specifications of instruction formats that human could easily understand and check.

Consistency is essential for verification of machine code because it guarantees that manipulation and reasoning over the abstract syntax of instructions can be mirrored precisely onto their binary forms. For example, verification of assemblers requires that instruction decoding reverts the assembling (encoding) process [20]. However, the previously proposed approaches to verifying instruction encoders and decoders all fail to establish consistency: to handle the complexity of instruction formats (especially that of CISC architectures), they employ expressive but ambiguous specifications such as context-free grammars or variants of regular expressions, from which it is impossible to derive consistent encoders and decoders. A representative example is the bidirectional grammar proposed by Tan and Morrisett [18]. It is an extension of regular expressions for writing instruction specifications from which verified encoders and decoders can be generated. However, because of the ambiguity of such specifications, two different abstract instructions may be encoded into the same *bit string* (i.e., a sequence of bits). When the decoder is deterministic, not all encoded instructions can be decoded back to the original instructions.

In this paper, we present an approach to automatic construction of instruction encoders and decoders that are verified to be consistent and sound. It is based on the observation that an instruction format inherently implies a bijection between abstract instructions and their binary forms that manifests as the determinacy of instruction decoding in actual hardware. This is true even for the most complicated CISC architectures. From a well-designed instruction specification that *precisely* captures this bijection, we are able to extract an appropriate representation of instructions, a pair of instruction encoder and decoder between this representation and the binary forms of instructions, and the consistency and soundness proofs of the encoder and decoder.

Based on the above ideas, we develop a framework for automatically generating consistent and sound instruction encoders and decoders. It extends the approach to specifying and generating instruction encoders and decoders proposed by Ramsey and Fernández [15] with mechanisms for *validating* their soundness and consistency by using theorem provers and SMT solvers. The framework consists of the following components (which are also our technical contributions):

- *A specification language for describing instruction formats.* This language is deliberately weaker in expressiveness than regular expressions while strong enough for describing instruction formats on common ISAs. Different from the existing ISA specification languages, it is rich enough for precisely capturing the syntactical structures of instructions and their operands, which implicitly encode a bijection between the abstract and the binary representations of instructions.
- *The algorithms for automatically generating encoders and decoders from instruction specifications.* Given any instruction specification, they generate an abstract syntax of instructions, a partial function from the abstract syntax to bit strings (i.e., an encoder) and a partial function from bit strings to the abstract syntax (i.e., a decoder). The generated definitions are formalized in

the Coq theorem prover so that the encoder and decoder can be formally validated later.

– *The algorithms for automatically validating the consistency and soundness of the generated encoders and decoders.* Given any instruction specification, they synthesize the consistency and soundness proofs for the generated encoder and decoder in Coq. This is possible because the bijection implied by the original specification guarantees that the encoder and decoder are inverses of each other, under the requirement that the binary "shapes" of different instructions or operands do not overlap with each other. This requirement is inherently satisfied by any instruction format, and can be easily proved with SMT solvers.

To demonstrate the effectiveness of our framework, we have applied it to a subset of 32-bit X86 instructions. In the rest of this paper, we first introduce relevant background information for this work and discuss the inadequacy of the existing work in Sect. 2. We then give an overview of our framework in Sect. 3 by further elaborating on the points above. After that, we discuss the definition of our specification language and the ideas supporting its design in Sect. 4. In the two subsequent sections Sect. 5 and Sect. 6, we discuss the algorithms for automatically generating and validating encoders and decoders. In Sect. 7, we present the evaluation of our framework. Finally, we discuss related work and conclude in Sect. 8.

## 2 Background

For our approach to work, the specification language we use must support the instruction formats on contemporary RISC and CISC architectures. In this section, we first introduce the key characteristics of these formats and then present a running example. We conclude this section by exposing the inadequacy of the existing approaches in capturing the bijections between the abstract and binary forms of instructions.

### 2.1 The Characteristics of Instruction Formats

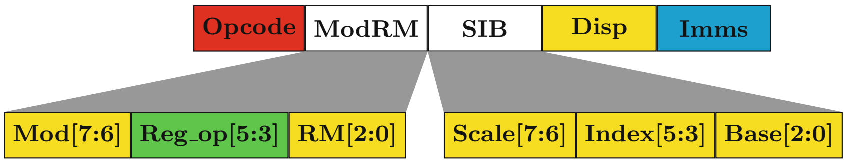


**Fig. 1.** The format of 32-bit X86 instructions

Instruction formats on CISC architectures may vary in length and structure even for the same type of instructions and may contain complex dependencies between their operands. In contrast, instructions on RISC architectures usually have fixed formats which are largely subsumed by CISC formats. Therefore, we focus on handling CSIC formats in this paper.

We use the format of 32-bit X86 instructions as an example to illustrate the complex characteristics of CISC instructions. It is depicted in Fig. 1. An instruction is divided into a sequence of *tokens* where each token is one or more bytes playing a particular role. The first token **Opcode** partially or fully determines the basic type of the instruction; it may be one to three bytes long. Following **Opcode** is an one-byte token **ModRM**. **ModRM** is further divided into a sequence of *fields* where a field $f[n_1 : n_2]$ represents a segment of the token named $f$ that occupies the $n_2$-th to $n_1$-th bits in that token. Depending on the value of **Opcode**, **ModRM** may or may not exist. When it exists, the value of **Reg_op[5:3]** may contain the encoded representation of a register operand. Another operand of the instruction may be an *addressing mode*. It is collectively determined by the values of **Mod[7:6]**, **RM[2:0]**, the token **SIB** (scaled index byte) and the displacement **Disp** following **ModRM**. Finally, the instruction may have an operand of immediate values in the token **Imms**.

For simplicity of our discussion, we have omitted some details such as the optional prefixes of instructions in Fig. 1. However, this simplified form is already enough to expose the key characteristics and complexity of CISC instruction formats (some of which also manifest in RISC). We summarize them below:

1. *Instructions as Composition of Components*: At the abstract level, an instruction consists of a collection of *components*. Each component serves a specific purpose and concretely corresponds to certain fields or tokens in the instruction format. For example, the constituents of 32-bit X86 instructions can be classified into four different kinds of components (marked with different colors in Fig. 1): the component determining the types of instructions (**Opcode**), the component denoting register operands (**Reg_op[5:3]**), the component denoting addressing modes (**Mod[7:6]**, **RM[2:0]**, **SIB** and **Disp**) and the component denoting immediate values (**Imms**).
2. *Variance of Components*: The concrete forms of components vary in different ways. A component may correspond to a single token (e.g., **Opcode** and **Imms**), a single field (e.g., **Reg_op[5:3]**), a mixing of fields and tokens (e.g., addressing modes), or other forms not shown here. Moreover, the abstract and concrete forms of a *single* type of components can also vary significantly such as the different addressing modes supported by X86 (as we shall see in detail in the following section).
3. *Interleaving of Components*. In most cases, there are clear sequential orders between the concrete representations of components. For example, the component of addressing modes immediately follows that of opcode and precedes that of immediate values. In the other cases, components may be interleaved with each other. For example, the component of register operands is interleaved with the component of addressing modes.
4. *Dependencies between and in Components*: The existence and forms of components are affected by the dependencies between each other and between their

own fields or tokens. For example, if an instruction does not take any argument, then the value of its **Opcode** determines that there is no token following **Opcode**. For another example, when **Mod**[**7:6**] contains the value `0b11`, the addressing mode is simply a register operand. Otherwise, the addressing mode may further depends on the values in **SIB** and **Disp**.

Note that, despite the above complexity, an instruction format is designed to inherently embed a (partial) bijection between the binary forms of instructions and their abstract representation as the composition of components. This is to ensure the determinacy of instruction decoding in hardware. This bijection is the central property to be investigated in this work.

## 2.2 A Running Example

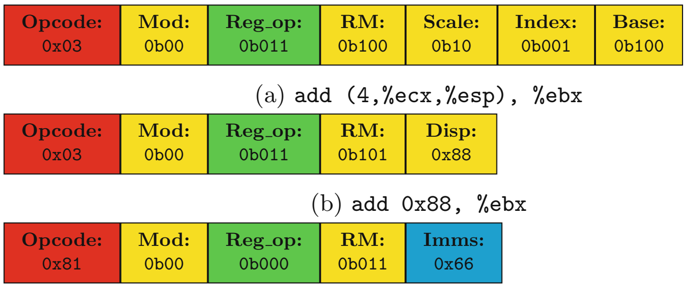
**Table 1.** The different forms of addressing modes

| AddrMode | Mod | RM | Scale | Index | Base | Disp |
|---|---|---|---|---|---|---|
| $\mathbf{r}$ | `0b11` | $\mathbf{r}$ | – | – | – | – |
| $(\mathbf{r})$ | `0b00` | $\mathbf{r} \neq$ `0b100` $\wedge \mathbf{r} \neq$ `0b101` | – | – | – | – |
| $(\mathbf{d})$ | `0b00` | `0b101` | – | – | – | $\mathbf{d}$ |
| $(\mathbf{s} * \mathbf{i} + \mathbf{b})$ | `0b00` | `0b100` | $\mathbf{s}$ | $\mathbf{i} \neq$ `0b100` | $\mathbf{b} \neq$ `0b101` | – |
| ... | ... | ... | ... | ... | ... | ... |

We present an example of encoding the `add` instruction to concretely illustrate the characteristics of the X86 instruction format. It will be used as a running example for the rest of the paper. The operands of `add` may have many forms. For simplicity, we only consider two cases: *1)* the first operand is a register while the second one is an addressing mode, and *2)* the first operand is an addressing mode while the second one is an immediate value.

In the first case, **Opcode** is `0x03`, indicating that **ModRM** exists and the first operand is encoded in its **Reg_op** field. The addressing mode has over 23 combinations because of the dependencies and constraints over their fields. We list only some of the combinations in Table 1, where - indicates that this field or token does not exist. The first row shows the direct addressing mode **r** where **Mod** is `0b11` and **RM** contains the encoded register operand **r**. The following three rows shows different kinds of indirect addressing modes. They are valid only if **Mod** is `0b00` and further constraints are satisfied. For example, the second row shows the indirect addressing mode **(r)** where **r** is encoded in **RM**. In this case, **r** must neither be **ESP** (encoded as `0b100`) nor be **EBP** (encoded as `0b101`). Similarly, the addressing mode $(\mathbf{s} * \mathbf{i} + \mathbf{b})$ requires that **RM** must be `0b100`, **Index** must not be `0b100` and **Base** must not be `0b101`.

In the second case, **Opcode** is `0x81`, indicating that **ModRM** exists, the first operand is an addressing mode, and the second operand is an immediate value following it. Here, **Reg_Op** must be `0b000`.



(a) `add (4,%ecx,%esp), %ebx`

(b) `add 0x88, %ebx`

(c) `add $0x66, (%ebx)`

**Fig. 2.** Some concrete examples of instruction encoding

We demonstrate the concrete examples of encoding `add (4,%ecx,%esp), %ebx`, `add 0x88, %ebx` and `add $0x66, (%ebx)` in Fig. 2 where `%ebx` and `%ecx` are encoded into `0b011` and `0b001`, respectively (the order of operands is *reversed* because we use the AT&T assembly syntax). Note how the forms of operands change significantly depending on the different values in the related fields. Note also, despite such complex dependencies, a bit string representing a valid `add` instruction corresponds to a *unique* combination of components.

## 2.3 Inadequacy of the Existing Approaches

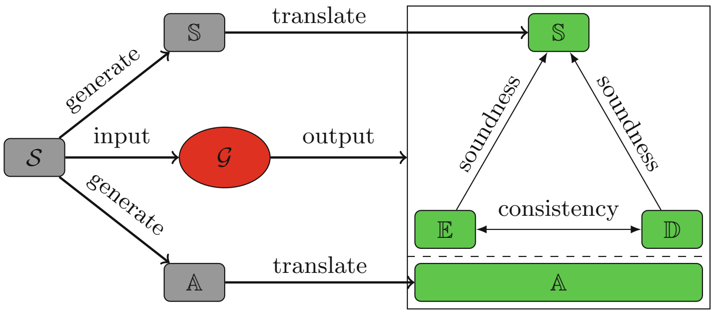
The existing approaches to specifying instructions are either *1)* too general and allow ambiguity or *2)* too low-level and break the component-based abstraction we just described. Either way, they fail to capture the inherent bijection embedded in an instruction format.

The bidirectional grammars [18] demonstrate the first kind of inadequacy. They contain the alternation grammar `Alt` $g_1$ $g_2$ for matching a bit string $s$ when either the sub-grammar $g_1$ or $g_2$ matches $s$. The ambiguity arises when both $g_1$ and $g_2$ match $s$: in this case, the same $s$ corresponds to two different internal representations. Therefore, bidirectional grammars cannot encode bijections in general. The same can be said for other work on verified parsing based on ambiguous grammars. We shall discuss them in detail in Sect. 8.

The Specification Language for Encoding and Decoding (or SLED) demonstrates the second kind of inadequacy [15]. It is a language for describing translations between symbolic and binary representations of machine instructions. On the surface, SLED takes the component-based view in specifying instructions. However, SLED specifications are interpreted through a normalization process by which every component is flattened into a sequence of tokens. After that, the structural information of components is completely lost. As a result, users can only derive encoders from the normalized specifications. They need to write decoders by using completely different specifications called "matching statements." This inability to generate matching encoders and decoders from a single specification is a common phenomenon in other approaches to ISA specifications.

In summary, no existing approach can precisely capture the bijections inherently embedded in instruction formats. This is the main intellectual problem we try to tackle in this paper. We shall elaborate on our solution to this problem in the remaining sections.

## 3 An Overview of the Framework



| | | |
|---|---|---|
| $\mathcal{S}$ : | instruction specifications | (in CSLED) |
| $\mathcal{G}$ : | algorithms for generating formal definitions and proofs | (in C++) |
| $\mathbb{A}$ : | abstract syntax of instructions | (on paper and in Coq) |
| $\mathbb{S}$ : | relational specifications of instructions | (on paper and in Coq) |
| $\mathbb{E}$ and $\mathbb{D}$ : | encoders and decoders | (in Coq) |

**Fig. 3.** The framework

We develop a framework for automatic generation of verified encoders and decoders that are consistent and sound. It is depicted in Fig. 3. To generate formally verified encoders and decoders, users first need to write down a specification of instructions $\mathcal{S}$ in a language called CSLED (or CoreSLED). CSLED is an enhancement to SLED for characterizing the bijection between the binary forms and the abstract syntax of instructions. Roughly speaking, $\mathcal{S}$ consists of a collection of *class* definitions, each of which defines a unique type of components that form instructions or their operands; the "top-most" class defines the type of instructions. Each class is associated with a set of *patterns* to uniquely determine a bijection between the binary and abstract forms of components in that class. Note that this bijection exists only when certain *well-formedness conditions* for patterns are satisfied. We shall elaborate on these ideas in Sect. 4.

From $\mathcal{S}$, the following definitions are generated and translated into Coq:

– The abstract syntax of instructions $\mathbb{A}$. It is a collection of algebraic data types corresponding to the classes defined in $\mathcal{S}$.
– A relational specification of $\mathcal{S}$ called $\mathbb{S}$. For each class, $\mathbb{S}$ contains a binary predicate that precisely captures the relation between components of that class and their binary forms. We write $\mathbb{R}[\![\mathcal{K}]\!]\ k\ l$ to denote that the component $k$ of class $\mathcal{K}$ has the binary form $l$.

Then, $\mathcal{S}$ is fed into a collection of algorithms $\mathcal{G}$ to generate the following definitions and proofs in Coq:

- An encoder $\mathbb{E}$ and a decoder $\mathbb{D}$. The encoder is a set of partial functions—one for each class—from the abstract syntax of that class to bit strings. We write $\mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor$ to denote that $l$ is the result of encoding a component $k$ of class $\mathcal{K}$ where $\lfloor\rfloor$ denotes the `some` constructor of the option type. Conversely, the decoder is a set of partial functions from bit strings to the abstract syntax. We write $\mathbb{D}_{\mathcal{K}}(l{+}{+}l') = \lfloor (k, l') \rfloor$ to denote the decoding of the bit string $l$ into a component $k$ of class $\mathcal{K}$ where $++$ is the append operation of bit strings. Here, the tailing bit string $l'$ represents the remaining bits after decoding the first component.
- The proof of consistency between the encoder and decoder. The consistency theorems are stated as the mutual inversion between the encoder and decoder:

$$\forall\ \mathcal{K}\ k\ l\ l', \mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor \Longrightarrow \mathbb{D}_{\mathcal{K}}(l{+}{+}l') = \lfloor (k, l') \rfloor.$$
$$\forall\ \mathcal{K}\ k\ l\ l', \mathbb{D}_{\mathcal{K}}(l{+}{+}l') = \lfloor (k, l') \rfloor \Longrightarrow \mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor.$$

  Their Coq proofs are automatically generated by inspecting the logical structure of classes and patterns in $\mathcal{S}$. For this, we need to derive a very important property: the decoder always decodes a bit string $l$ back to the same sequence of components. We achieve this goal by combining proofs in Coq with SMT solving of verification conditions that are automatically derived from well-formed specifications.
- The proof of soundness of the encoder and decoder. The soundness theorems are stated as follows:

$$\forall\ \mathcal{K}\ k\ l\ l', \mathbb{E}_{\mathcal{K}}(k) = \lfloor l \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ l.$$
$$\forall\ \mathcal{K}\ k\ l\ l', \mathbb{D}_{\mathcal{K}}(l{+}{+}l') = \lfloor (k, l') \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ l.$$

  As we shall see later, $\mathbb{E}_{\mathcal{K}}$ and $\mathbb{R}[\![\mathcal{K}]\!]$ are both defined recursively on the definition of classes in $\mathcal{S}$. Their main difference is that the former is a function while the latter is a relation. Therefore, it is easy to prove the first soundness theorem by induction on $k$. By using the second consistency theorem and the first soundness theorem, we can easily prove the second soundness theorem.

As we shall see in the following sections, the actual implementations of encoders and decoders and their consistency and soundness theorems are more complicated than presented here. Nevertheless, the above discussion covers the high-level ideas of our framework.

Note that in Fig. 3, $\mathcal{S}$ and $\mathcal{G}$ are not formalized and hence not in the trusted base. The consistency and soundness of $\mathbb{E}$ and $\mathbb{D}$ are independently *validated* by using Coq and SMT solvers. If the validation of either property fails, the framework reports a failed attempt to generate the encoder and decoder. This often indicates that the instruction specification is not well-formed.

## 4 The Specification Language

The key idea underlying the design of CSLED is to record explicitly the structures of components in instruction specifications, instead of normalizing them into tokens as did in SLED. In this way, CSLED specifications accurately capture the key characteristics of instruction formats described in Sect. 2.1, hence the bijections embedded in instruction formats. In this section, we present the syntax of CSLED, explain the ideas underlying its design, and use the running example to illustrate how CSLED specifications are written. We also introduce the syntactical and relational interpretations of CSLED specifications and present the well-formedness conditions for the bijections to exist.

### 4.1 The Syntax

$$
\begin{array}{rl}
\mathcal{S} ::= & \langle \texttt{empty} \rangle \\
 & \mid \mathcal{S}\,\mathcal{D} \\
 & \\
\mathcal{D} ::= & \texttt{token}\ \mathit{tid} = \mathcal{T}; \\
 & \mid \texttt{field}\ \mathit{fid} = \mathcal{F}; \\
 & \mid \texttt{class}\ \mathit{kid} = \mathcal{K}; \\
 & \\
\mathcal{T} ::= & \texttt{(}n\texttt{)} \\
\mathcal{F} ::= & \mathit{tid}\texttt{(}n_1 : n_2\texttt{)} \\
\mathcal{K} ::= & \mathcal{B} \\
 & \mid \mathcal{K}\ \texttt{|}\ \mathcal{B} \\
\mathcal{B} ::= & \texttt{constr}\ \mathit{cid}\ [\mathit{aid}]\ (\mathcal{P})
\end{array}
$$

(a) Definitions

$$
\begin{array}{rl}
\mathcal{P} ::= & \mathcal{J} \\
 & \mid \mathcal{P}\,\texttt{;}\,\mathcal{J} \\
\mathcal{J} ::= & \mathcal{A} \\
 & \mid \mathcal{J}\texttt{\&}\mathcal{A} \\
\mathcal{A} ::= & \mathcal{O} \\
 & \mid \texttt{cls}\ \%i \\
\mathcal{O} ::= & \epsilon : \mathit{tid} \\
 & \mid \mathit{fid} = n \\
 & \mid \mathit{fid} \neq n \\
 & \mid \texttt{fld}\ \%i \\
 & \mid \mathcal{O}\ \texttt{\&}\ \mathcal{O} \\
 & \mid \mathcal{O}\ \texttt{;}\ \mathcal{O}
\end{array}
$$

(b) Patterns

**Fig. 4.** The syntax of CSLED

The syntax of CSLED is shown in Fig. 4. A CSLED specification (denoted by $\mathcal{S}$) consists of a list of *definitions* (denoted by $\mathcal{D}$). The three kinds of definitions are for tokens (denoted by $\mathcal{T}$), fields (denoted by $\mathcal{F}$) and classes (denoted by $\mathcal{K}$). Every definition is bound to a unique identifier where *tid*, *fid* and *kid* represents the identifiers of tokens, fields and classes, respectively.

Tokens represent consecutive segments of bytes and are the basic elements for forming instructions. They are necessary for distinguishing the same sequence of bytes with different interpretations. Their definitions have the form $(n)$ where $n$ must be divisible by 8 which denotes a token of $n$-bits or $n/8$ bytes. Definitions of fields have the form $\mathit{tid}\texttt{(}n_1 : n_2\texttt{)}$ which denotes a field occupying the $n_2$-th to $n_1$-th bits in the token *tid*.

Classes represent specific types of components. They play a central role in the specifications by accurately capturing the component-based abstraction we discussed in Sect. 2.1. A class consists of a collection of *branches* (denoted by $\mathcal{B}$) each of which denotes a possible form of components in the class. Definitions of branches have the form `constr` *cid* $[aid]$ $(\mathcal{P})$ where *cid* is a unique identifier for the branch (denoting a constructor) and $[aid]$ is a list of *fid* or *kid* denoting the sub-components or fields for constructing a component (i.e., the arguments to the constructor). These arguments capture the nested structures of components where a bigger component may be constructed from smaller ones or basic fields.

A branch is associated with a single *pattern* $\mathcal{P}$. A pattern plays two roles: it determines the types of a sequence of tokens that concretely forms components of this branch, and it describes a relation between these tokens (and their fields) with the abstract arguments of the branch. This relation essentially encodes the bijection between the abstract and binary forms of components in this branch.

At the top-most level, $\mathcal{P}$ is a sequence of *judgments* (denoted by $\mathcal{J}$) separated by `;`, such that $\mathcal{J}_1\,\texttt{;}\,\ldots\,\texttt{;}\,\mathcal{J}_n$ matches a sequence of tokens concretely represented by a bit string $l$ if and only if $l = l_1{+}{+}l_2{+}{+}\ldots{+}{+}l_n$ and $\mathcal{J}_i$ matches $l_i$ for $1 \leq i \leq n$. This sequential pattern is enough for relating abstract and binary forms of components when each $\mathcal{J}_i$ (and $l_i$) corresponds to a single (sub-)component. However, according to the discussion in Sect. 2.1, components may be interleaved with each other and $\mathcal{J}_i$ may correspond to multiple components. Therefore, a judgment is a conjunction of *atomic patterns* (denoted by $\mathcal{A}$) each of which matches an interleaved component. In case there is no interleaving, a judgment reduces to a single atomic pattern.

An atomic pattern has two forms: `cls` $\%i$ for relating a sequence of tokens to the $i$-th argument in $[aid]$ of the corresponding branch which must be a class, and $\mathcal{O}$ for relating tokens to field arguments in $[aid]$ and for further constraining the fields of these tokens. The $\mathcal{O}$ patterns are called *basic patterns*. Among them $\epsilon\texttt{:}\,tid$ matches any token of type *tid*; $fid = n$ ($fid \neq n$) matches a token with the field *fid* whose value is (is not) the constant $n$; similar to `cls` $\%i$, `fld` $\%i$ relates the $i$-th argument in $[aid]$ of the branch which must be a field to the concrete value of the field in the matching token. The last two cases of basic patterns indicate that arbitrary sequencing and interleaving of basic patterns are allowed. Despite such free interleaving, a basic pattern can only match with sequences of tokens of the same length and of a unique type because we require that $\mathcal{O}_1$ `&` $\mathcal{O}_2$ be well-formed only if both $\mathcal{O}_1$ and $\mathcal{O}_2$ match sequences of tokens with the same type. Therefore, basic patterns have the same expressiveness as SLED specifications in their normalized forms [15].

In contrast to basic patterns, judgments and atomic patterns are much more expressive as they may match tokens of different lengths and forms. This is because a class pattern `cls` $\%i$ can match components of a class $\mathcal{K}$ with multiple branches, each of which may have different patterns. By introducing class patterns into atomic patterns, we are able to represent the complete structures of components and establish bijections from these structures. This is the key improvement we made in CSLED compared to SLED.

## 4.2 The CSLED Specification of the Running Example

```
token Opcode = (8);     token Disp = (32);     token Imms = (32);
token ModRM = (8);      token SIB = (8);

field opcode = Opcode(7 : 0);       field disp = Disp(31 : 0);
field imms = Imms(31 : 0);          field mod = ModRM(7 : 6);
field reg_op = ModRM(5 : 3);        field rm = ModRM(2 : 0);
field scale = SIB(7 : 6);           field index = SIB(5 : 3);
field base = SIB(2 : 0);

class Addrmode  =
| constr addr_r [rm]  (mod = 0b11 & fld %1)
| constr addr_ir [rm]  (mod = 0b00 & rm ≠ 0b100 & rm ≠ 0b101 & fld %1)
| constr addr_disp [disp]  (mod = 0b00 & rm = 0b101; fld %1)
| constr addr_sib [scale, index, base]
   (mod = 0b00 & rm = 0b100;
   fld %1 & fld %2 & fld %3 & index ≠ 0b100 & base ≠ 0b101)
...

class Instruction  =
| constr AddGvEv [reg_op, Addrmode]  (opcode = 0x03; fld %1 & cls %2)
| constr AddEvIz [Addrmode, imms]
   (opcode = 0x81; reg_op = 0b000 & cls %1; fld %2)
...
```

**Fig. 5.** The CSLED specification of the running example

The CSLED specification of our running example is depicted in Fig. 5. The *Addrmode* class specifies the possible addressing modes. Its branches are translated from the addressing modes described in Table 1 one by one, such that their patterns exactly match the binary structures of components in the corresponding branches. For instance, the branch *addr_sib* is translated from the fourth addressing mode in Table 1. Its pattern is a sequence of two judgment. The first judgment is a conjunction of two basic patterns that are the required constraints on the fields *mod* and *rm* of *ModRM* described in Table 1. Therefore, it must match the single token *ModRM*. The second judgment is a conjunction of basic patterns that constrain the fields *index* and *base* of *SIB* and relate arguments of *addr_sib* with the concrete values in the fields *scale*, *index* and *base*. Because these patterns all constrain the fields of *SIB*, the second judgment must match the single token *SIB*.

Similarly, the *Instruction* class specifies the instructions. Its two branches characterize the two kinds of `add` instructions described in Sect. 2.2. Note how conjunctions between the basic patterns for *reg_op* and class patterns for *Addrmode* are used to describe the interleaving of register operands and addressing modes. Note also that in every branch of *Addrmode* the first pattern matches the token *ModRM*, and in any branch of *Instruction* the token *Opcode* is always followed by *Addrmode*. Therefore, *ModRM* always follows *Opcode* as desired.

By this example, we demonstrate the critical feature of CSLED: because the syntax of CSLED is designed to precisely describe instruction formats in ISA manuals, it implicitly captures the embedded bijections. Note that, because of its faithfulness to the ISA manuals, CSLED's syntax contains full details about instruction encoding by nature. However, it is not hard to imagine this syntax being refined to the client's syntax through another straightforward bijection. In fact, this is how we anticipate clients will use CSLED in practice, e.g., to build verified assemblers for X86.

## 4.3 Interpretation of CSLED Specifications

From a CSLED specification $\mathcal{S}$, we extract *1)* a collection of data types for representing the abstract syntax of components, and *2)* a collection of binary relations between these data types and bit strings for representing the mappings between the abstract and concrete forms of components.

**Data Types of Components.** We use the operator $\mathbb{T}[\![-]\!]$ to denote the interpretation of basic fields and classes into data types. The translation for fields are simple: given a field definition `field` $\mathit{fid} = \mathit{tid}\texttt{(}n_1 : n_2\texttt{)}$, $\mathbb{T}[\![\mathit{fid}]\!] = \langle n_1 - n_2 + 1\rangle$ where $\langle n\rangle$ represent an unsigned binary integer of $n$ bits. Note that we do not further translate the values of fields as they have straightforward interpretations (such as the mapping from bits to registers described in Sect. 2.1). The interpretation of classes is only slightly more involved. Given a class definition `class` $\mathit{kid} = \mathcal{K}$, $\mathbb{T}[\![\mathit{kid}]\!]$ is an algebraic data type named *kid*. For each branch `constr` $\mathit{cid}\ [\mathit{aid}_1, \ldots, \mathit{aid}_n]\ \mathcal{P}$ of $\mathcal{K}$, there is a constructor *cid* for *kid* that takes $n$ arguments of types $\mathbb{T}[\![\mathit{aid}_1]\!], \ldots, \mathbb{T}[\![\mathit{aid}_n]\!]$.

**Relations Derived from CSLED.** The translation of CSLED specifications into relations is defined in Fig. 6. Here, *BS* denotes the type of bit strings. When $\mathit{aids} = [\mathit{aid}_1, \ldots, \mathit{aid}_n]$ we write $\mathbb{T}[\![\mathit{aids}]\!]$ to denote the product type of $\mathbb{T}[\![\mathit{aid}_1]\!], \ldots, \mathbb{T}[\![\mathit{aid}_n]\!]$. We use $\equiv$ to denote the definitional equality.

The function $\mathbb{R}[\![\mathit{aid}]\!]$ translates a type of components associated with *aid* into a binary relation between its abstract representation and bit strings, where *aid* may denote a field or a class. The definition for field components is straightforward. $\mathbb{R}[\![\mathit{kid}]\!]\ k\ l$ holds iff there is a branch of *kid* whose interpretation relates $k$ and $l$, which further requires (by the third rule in Fig. 6) that $k$ is constructed by using the constructor of that branch and the pattern of the branch relates the arguments of the constructor to $l$. The latter relation is defined by $\mathbb{R}_p[\![-,-]\!]$ such that $\mathbb{R}_p[\![\mathcal{P}, \mathit{aids}]\!]\ \mathit{args}\ l$ holds iff $\mathcal{P}$ matches $l$ and the arguments *args* satisfy the constraints enforced by $\mathcal{P}$ and *aids*. More specifically, $\mathbb{R}_p[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, \mathit{aids}]\!]\ \mathit{args}\ l$ holds iff $\mathcal{P}$ matches a prefix of $l$ and $\mathcal{J}$ matches the rest of $l$. The definition of $\mathbb{R}_p[\![\mathcal{J}\,\texttt{\&}\,\mathcal{A}]\!]$ is slightly different in that $\mathbb{R}_p[\![\mathcal{J}\,\texttt{\&}\,\mathcal{A}, \mathit{aids}]\!]\ \mathit{args}\ l$ holds iff $\mathcal{A}$ matches the whole $l$ and $\mathcal{J}$ matches a prefix of $l$. This is necessary for describing the

$$
\begin{aligned}
\mathbb{R}[\![\mathit{fid}]\!] &::= \lambda(f : \mathbb{T}[\![\mathit{fid}]\!])\ (l : \mathit{BS}).\\
&\quad \exists(\mathit{tid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv \texttt{(}n_3\texttt{)} \land \mathit{fid} \equiv \mathit{tid}\texttt{(}n_1 : n_2\texttt{)}\\
&\quad \land \mathit{length}(l) = n_3 \land l[n_1 : n_2] = f\\
\mathbb{R}[\![\mathit{kid}]\!] &::= \lambda(k : \mathbb{T}[\![\mathit{kid}]\!])\ (l : \mathit{BS}).\\
&\quad \exists \mathcal{B}, \mathit{kid} \equiv \ldots \texttt{|} \mathcal{B} \texttt{|} \ldots \land \mathbb{R}_b[\![\mathcal{B}, \mathit{kid}]\!]\ k\ l\\
\mathbb{R}_b[\![\mathcal{B}, \mathit{kid}]\!] &::= \lambda(k : \mathbb{T}[\![\mathit{kid}]\!])\ (l : \mathit{BS}).\\
&\quad \exists \mathit{args}, k = \mathit{cid}\ \mathit{args} \land \mathbb{R}_p[\![\mathcal{P}, \mathit{aids}]\!]\ \mathit{args}\ l\\
&\quad (\text{where } \mathcal{B} = \texttt{constr}\ \mathit{cid}\ \mathit{aids}\ \mathcal{P})\\
\mathbb{R}_p[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\exists l_1\ l_2, l = l_1 +\!\!+ l_2\\
&\quad \land \mathbb{R}_p[\![\mathcal{P}, \mathit{aids}]\!]\ \mathit{args}\ l_1 \land \mathbb{R}_p[\![\mathcal{J}, \mathit{aids}]\!]\ \mathit{args}\ l_2\\
\mathbb{R}_p[\![\mathcal{J}\,\texttt{\&}\,\mathcal{A}, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\exists l_1\ l_2, l = l_1 +\!\!+ l_2\\
&\quad \land \mathbb{R}_p[\![\mathcal{J}, \mathit{aids}]\!]\ \mathit{args}\ l_1 \land \mathbb{R}_p[\![\mathcal{A}, \mathit{aids}]\!]\ \mathit{args}\ l\\
\mathbb{R}_p[\![\epsilon\,\texttt{:}\,\mathit{tid}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\exists n, \mathit{tid} \equiv \texttt{(}n\texttt{)} \land \mathit{length}(l) = n\\
\mathbb{R}_p[\![\mathit{fid} = n, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS})\ .\exists(\mathit{tid}\ \mathit{fid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv \texttt{(}n_3\texttt{)}\\
&\quad \land \mathit{fid} \equiv \mathit{tid}\texttt{(}n_1 : n_2\texttt{)} \land \mathit{length}(l) = n_3 \land l[n_1 : n_2] = n\\
\mathbb{R}_p[\![\mathit{fid} \neq n, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS})\ .\exists(\mathit{tid}\ \mathit{fid}\ n_1\ n_2\ n_3), \mathit{tid} \equiv \texttt{(}n_3\texttt{)}\\
&\quad \land \mathit{fid} \equiv \mathit{tid}\texttt{(}n_1 : n_2\texttt{)} \land \mathit{length}(l) = n_3 \land l[n_1 : n_2] \neq n\\
\mathbb{R}_p[\![\texttt{fld}\ \%i, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\mathbb{R}[\![\mathit{aids}[i]]\!]\ \mathit{args}[i]\ l\\
\mathbb{R}_p[\![\texttt{cls}\ \%i, \mathit{aids}]\!] &::= \lambda(\mathit{args} : \mathbb{T}[\![\mathit{aids}]\!])\ (l : \mathit{BS}).\mathbb{R}[\![\mathit{aids}[i]]\!]\ \mathit{args}[i]\ l
\end{aligned}
$$

**Fig. 6.** Translation of CSLED specifications into relations

interleaving of components. Furthermore, certain constraints need to be satisfied for deriving a bijection as shall discuss in Sect. 4.4. $\mathbb{R}_p[\![\mathcal{O}_1\,\texttt{;}\,\mathcal{O}_2, \mathit{aids}]\!]$ and $\mathbb{R}_p[\![\mathcal{O}_1\texttt{\&}\mathcal{O}_2, \mathit{aids}]\!]$ are not shown in Fig. 6 because they are defined the same as $\mathbb{R}_p[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, \mathit{aids}]\!]$ and $\mathbb{R}_p[\![\mathcal{J}\texttt{\&}\mathcal{A}, \mathit{aids}]\!]$, respectively. $\mathbb{R}_p[\![\mathit{fid} = n, \mathit{aids}]\!]\ \mathit{args}\ l$ holds iff $l$ is a token containing *fid* whose value is $n$; similar for $\mathbb{R}_p[\![\mathit{fid} \neq n, \mathit{aids}]\!]$. $\mathbb{R}_p[\![\texttt{fld}\ \%i, \mathit{aids}]\!]$ holds iff the $i$-th argument in *args* matches with the concrete value found in $l$; same for $\mathbb{R}_p[\![\texttt{cls}\ \%i, \mathit{aids}]\!]$. Note how the last two definitions make use of *args* for getting the values of arguments.

## 4.4 Well-Formedness of Specifications

The binary relation we define in the last section denotes a bijection only when the CSLED specification under investigation satisfies certain well-formedness conditions. These conditions guarantee that, given any bit string $l$, there is at most one abstract object related to $l$ via the defined binary relation. Well-formedness is the composition of three properties which we call *disjointness*, *compatibility*, and *uniqueness*. We give and explain their definitions below. The logic for checking these conditions is embedded in the generation algorithms we will discuss in the next section and will be exploited for the validation of the generated encoders and decoders.

**Disjointness.** Given a pattern $\mathcal{P}_1 \& \mathcal{P}_2$, it satisfies disjointness if $\mathcal{P}_1$ and $\mathcal{P}_2$ match disjoint fields.[1] To understand this, suppose $\mathcal{P}_1$ and $\mathcal{P}_2$ relate different abstract arguments $a_1$ and $a_2$ to overlapping bits in a bit string $l$. Then, we cannot determine if the values in the overlapping bits are for $a_1$ or $a_2$. Hence, the derived binary relation cannot possibly be a bijection. Disjointness rules out such possibility.

**Compatibility.** We call the types of sequences of tokens a pattern $\mathcal{P}$ matches the "shapes" of $\mathcal{P}$. Given a pattern $\mathcal{P}_1 \& \mathcal{P}_2$, it satisfies compatibility if every possible shape of $\mathcal{P}_1$ is in a prefix of every possible shape of $\mathcal{P}_2$ when $\mathcal{P}_2$ is a class pattern (and vice versa). Enforcing compatibility simplifies the interpretation of $\mathcal{P}_1 \& \mathcal{P}_2$ when $\mathcal{P}_1$ or $\mathcal{P}_2$ is a class pattern with multiple branches that may match bit strings with different shapes. Compatibility makes sense because for common instruction formats it is always the case that the components matched by $\mathcal{P}_1$ are embedded in the *longest common prefixes* of all the possible shapes of $\mathcal{P}_2$ when $\mathcal{P}_2$ is a class pattern (and vice versa). For example, in the example depicted in Fig. 2, **Reg_op** is always embedded into the common prefix of all the possible shapes of addressing modes, i.e., the **ModRM** token.

**Uniqueness.** Given a class pattern $\mathcal{K}$, it satisfies uniqueness if for any bit string $l$, at most one of its branches matches $l$. Uniqueness is essential for ensuring the determinacy of decoders in presences of class patterns. Fortunately, it implicitly holds for common instruction formats as they are designed with determinacy of decoding in mind. To concretely check the uniqueness implied by instruction formats, we first define the *structural condition* for a branch with pattern $\mathcal{P}$ as the conjunction of the statically known constraints in $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]_{cond}$. We then require that no structure conditions for any two branches of a class can be satisfied simultaneously. This requirement allows us to uniquely determine the branch used to construct a class component. For example, the structural conditions of the first three branches of *Addrmode* are ($mod = \texttt{0b11}$), ($mod = \texttt{0b00} \;\&\; rm \neq \texttt{0b100} \;\&\; rm \neq \texttt{0b101}$) and ($mod = \texttt{0b00} \;\&\; rm = \texttt{0b101}$). Obviously, any pairwise combination of these conditions cannot possibly be satisfied. This is true even if we consider all the branches of *Addrmode*. Therefore, there is at most one way to decode any addressing mode.

## 5 Generation of Encoders and Decoders

We discuss the algorithm for generating encoders and decoders from CSLED specifications. The structures of these encoders and decoders closely match the relations derived from specifications. Furthermore, every operation in an encoder has a counterpart in the corresponding decoder, and vice versa.

[1] We abuse the notation by using $\mathcal{P}$ to denote suitable patterns such as $\mathcal{J}$, $\mathcal{A}$ or $\mathcal{O}$.

## 5.1 Generation of Encoders

$$
\begin{aligned}
\mathcal{G}_{\mathbb{E}}[\![\epsilon : tid, bs, args]\!] &::= \lfloor bs \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![fid = n, bs, args]\!] &::= write_{fid}\ bs\ n \\
\mathcal{G}_{\mathbb{E}}[\![fid \neq n, bs, args]\!] &::= assert(read_{fid}\ bs \neq n) \\
\mathcal{G}_{\mathbb{E}}[\![\texttt{fld}\ \%i, bs, args]\!] &::= write_{fid}\ bs\ args[i] \qquad \text{(where } fid \text{ is the field id of } args[i]\text{)} \\
\mathcal{G}_{\mathbb{E}}[\![\texttt{cls}\ \%i, bs, args]\!] &::= \mathbb{E}_{\mathcal{K}}(args[i], bs) \qquad \text{(where } \mathcal{K} \text{ is the class of } args[i]\text{)} \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{;}\ \mathcal{O}_2, bs, args]\!] &::= l_1 \leftarrow first\_n(bs, [\![\mathcal{O}_1]\!]_{tokens}); l_2 \leftarrow skip\_n(bs, [\![\mathcal{O}_1]\!]_{tokens}) \\
&\phantom{::=}\ bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1, l_1, args]\!]; bs_2 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_2, l_2, args]\!]; \lfloor bs_1 +\!\!+ bs_2 \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{\&}\ \mathcal{O}_2, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1, bs, args]\!]; \mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_2, bs_1, args]\!] \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]; bs' \leftarrow skip\_n(bs, |bs_1|); \\
&\phantom{::=}\ bs_2 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{J}, bs', args]\!]; \lfloor bs_1 +\!\!+ bs_2 \rfloor \\
\mathcal{G}_{\mathbb{E}}[\![\mathcal{J}\texttt{\&}\mathcal{A}, bs, args]\!] &::= bs_1 \leftarrow \mathcal{G}_{\mathbb{E}}[\![\mathcal{J}, bs, args]\!]; \mathcal{G}_{\mathbb{E}}[\![\mathcal{A}, bs_1, args]\!]
\end{aligned}
$$

**Fig. 7.** Generation of encoders from patterns

From every class $\mathcal{K}$, we extract an encoder $\mathbb{E}_{\mathcal{K}}$ for its components. It is a partial function that takes two arguments—a component $k$ and a bit string $l$ representing the result previously generated by encoders—and outputs an updated bit string if the encoding succeeds. We shall write $\mathbb{E}_{\mathcal{K}}(k, l) = \lfloor l' \rfloor$ to denote that $l'$ is the result of encoding $k$ on top of $l$.

$\mathbb{E}_{\mathcal{K}}(k, l)$ is defined by recursion on the structure of $k$. For every branch $\mathcal{B}$ of $\mathcal{K}$, we generate a piece of Coq code from the pattern $\mathcal{P}$ of $\mathcal{B}$ for encoding $k$. We then insert it into the definition of $\mathbb{E}_{\mathcal{K}}(k, l)$. We write $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$ to denote the code snippet so generated, where $bs$ is the name of the generated bit string at this point and $args$ contains the names of the arguments to the constructor. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$ is defined in Fig. 7 where we use the option monad for sequencing the encoding operations. The first case is obvious. Code generated by $\mathcal{G}_{\mathbb{E}}[\![fid = n, bs, args]\!]$ writes the constant $n$ into the field associated with $fid$. $\mathcal{G}_{\mathbb{E}}[\![fid \neq n, bs, args]\!]$ checks whether the corresponding field contains the constant $n$ and returns none if the checking fails. $\mathcal{G}_{\mathbb{E}}[\![\texttt{fld}\ \%i, bs, args]\!]$ writes the value of the $i$-th argument into the corresponding field. $\mathcal{G}_{\mathbb{E}}[\![\texttt{cls}\ \%i, bs, args]\!]$ calls the encoder for the class corresponding to $\texttt{cls}\ \%i$. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\ \texttt{;}\ \mathcal{O}_2, bs, args]\!]$ encodes its two parts recursively and concatenates the results together, where $first\_n(bs, n)$ returns the first $n$ bits in $bs$ and $skip\_n(bs, n)$ skips the first $n$ bits in $bs$ and returns the remaining ones. $\mathcal{G}_{\mathbb{E}}[\![\mathcal{O}_1\texttt{\&}\mathcal{O}_2, bs, args]\!]$ first encodes data matching $\mathcal{O}_1$, and then passes the result to the encoding for $\mathcal{O}_2$. The last two cases are similar. Note that if the generated code occurs at the beginning of a branch, then $bs$ coincides with the input argument $l$. Otherwise, $bs$ denotes intermediate results. As we can see, all these cases follow the logical structure of CLSED specifications we have described before.

## 5.2 Generation of Decoders

From every class $\mathcal{K}$, we extract a decoder $\mathbb{D}_{\mathcal{K}}$. It is a partial function such that $\mathbb{D}_{\mathcal{K}}(l) = \lfloor(k, l_1, l_2)\rfloor$ holds iff $l = l'{+}{+}l_2$, $l'$ is the binary representation of $k$, and $l_1$ is the result of inverting the encoding operation, i.e., setting every bit the decoder touches in $l'$ to 0. This extra return value is introduced to help with the verification as we shall see in Sect. 6.

$$
\begin{aligned}
\mathcal{G}_{\mathbb{D}}[\![\epsilon : tid, bs, args]\!] &::= remains \leftarrow skip\_n(bs, tid);\ \lfloor(bs,\ remains)\rfloor \\
\mathcal{G}_{\mathbb{D}}[\![fid = n, bs, args]\!] &::= ori \leftarrow clear_{fid}\ bs;\ remains \leftarrow skip\_n(bs, tid); \\
&\qquad \lfloor(ori,\ remains)\rfloor \qquad (\text{where } fid \equiv tid\texttt{(}n_1 : n_2\texttt{)}) \\
\mathcal{G}_{\mathbb{D}}[\![fid \neq n, bs, args]\!] &::= ori \leftarrow clear_{fid}\ bs;\ remains \leftarrow skip\_n(bs, tid); \\
&\qquad \lfloor(ori,\ remains)\rfloor \qquad (\text{where } fid \equiv tid\texttt{(}n_1 : n_2\texttt{)}) \\
\mathcal{G}_{\mathbb{D}}[\![\texttt{fld}\ \%i, bs, args]\!] &::= argi \leftarrow read_{fid}\ bs;\ ori \leftarrow clear_{fid}\ bs; \\
&\qquad remains \leftarrow skip\_n(bs, tid);\ \lfloor(ori,\ remains)\rfloor \\
&\qquad (\text{where } fid \text{ is the field id of } args[i]) \\
\mathcal{G}_{\mathbb{D}}[\![\texttt{cls}\ \%i, bs, args]\!] &::= argi, origin, remains \leftarrow \mathbb{D}_{\mathcal{K}}(bs);\ \lfloor(origin,\ remains)\rfloor \\
&\qquad (\text{where } \mathcal{K} \text{ is the class of } args[i]) \\
\mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_1\ \texttt{;}\ \mathcal{O}_2, bs, args]\!] &::= ori_1, remains_1 \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_1, bs, args]\!]; \\
&\qquad ori_2, remains_2 \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_2, remains_1, args]\!]; \\
&\qquad \lfloor(ori_1\ {+}{+} ori_2,\ remains_2)\rfloor \\
\mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_1\ \texttt{\&}\ \mathcal{O}_2, bs, args]\!] &::= remains \leftarrow skip\_n(bs, [\![\mathcal{O}_2]\!]_{tokens}); \\
&\qquad ori, \_ \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_2, bs, args]\!]; \\
&\qquad orilst, \_ \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{O}_1, ori, args]\!]; \\
&\qquad \lfloor(orilst,\ remains)\rfloor \\
\mathcal{G}_{\mathbb{D}}[\![\mathcal{P}\,\texttt{;}\,\mathcal{J}, bs, args]\!] &::= ori_1, remains_1 \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{P}, bs, args]\!]; \\
&\qquad ori_2, remains_2 \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{J}, remains_1, args]\!]; \\
&\qquad \lfloor(ori_1{+}{+}ori_2, remains_2)\rfloor \\
\mathcal{G}_{\mathbb{D}}[\![\mathcal{J}\texttt{\&}\mathcal{A}, bs, args]\!] &::= ori, remains \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{A}, bs, args]\!]; \\
&\qquad orilst, \_ \leftarrow \mathcal{G}_{\mathbb{D}}[\![\mathcal{J}, ori, args]\!]; \\
&\qquad \lfloor(orilst, remains)\rfloor
\end{aligned}
$$

**Fig. 8.** Generation of decoders from patterns

The first step of $\mathbb{D}_{\mathcal{K}}$ is to decide which branch of $\mathcal{K}$ should be chosen for decoding $l$. It can be done by checking the structural conditions derived from the patterns of branches (which we have introduced in Sect. 4.4) against $l$. Specifically, for the pattern $\mathcal{P}$ of each branch of $\mathcal{K}$, we translate its structural condition $[\![\mathcal{P}]\!]_{cond}$ into a decision procedure in Coq (a function returning boolean values) in a straightforward manner. We then insert an if-statement to check if $[\![\mathcal{P}]\!]_{cond}$ can be satisfied. If so, we start the decoding process for this branch. Otherwise, we repeatedly check other branches until a matching case is found. Note also that by uniqueness, there is at most one structural condition that can be satisfied. Therefore, $\mathbb{D}_{\mathcal{K}}$ is deterministic in choosing branches.

Once a matching branch is found, we use the algorithm $\mathcal{G}_{\mathbb{D}}[\![\mathcal{P}, bs, args]\!]$ (the counterpart of $\mathcal{G}_{\mathbb{E}}[\![\mathcal{P}, bs, args]\!]$) to generate a piece of Coq code for decoding the arguments of this branch. It is defined in Fig. 8. Similar to encoding, the generated code snippet follows the logical structure of CSLED specifications. The function $clear_{fid}\ bs$ set the bits of the field *fid* in *bs* to 0. Note that the decoding operations are exactly the inversion of those in Fig. 7. Note also that the fourth and fifth cases in Fig. 8 are responsible for decoding the arguments and storing them in *argi*. By applying the corresponding constructor to these arguments, we get the output component $k$, which together with the two values returned by $\mathcal{G}_{\mathbb{D}}$ form the final output of $\mathbb{D}_{\mathcal{K}}$ .

## 5.3 Generation for the Running Example

We show the representative cases of the generated encoder and decoder for our running example in Fig. 9. They include the encoding and decoding procedures for the fourth branch of *Addrmode* (the most complicated one). We can see that the encoding and decoding operations are exactly the inverses of each other. The encoder first writes the fields in *ModRM* and then those in *SIB*. Conversely, the decoder first reads the fields in *ModRM* and then those in *SIB*. Finally, it forms the component and returns the reverted and remaining bits. The function `BF_addr_sib` is the decision procedure generated from the structural condition for the fourth branch of *Addrmode*. We also show the encoding and decoding procedures for the first `add` instruction in Fig. 9. Their structures are very similar to those of *Addrmode*.

## 6 Validation of Encoders and Decoders

In this section, we discuss how to exploit the logical structure of and the well-formedness conditions for CSLED specifications to automatically synthesize the proofs of consistency and soundness for encoders and decoders.

### 6.1 Synthesizing the Proof of Consistency

The consistency between encoders and decoders is composed of two properties and stated as follows:

**Theorem 1 (Consistency between Encoders and Decoders).** *Given any class $\mathcal{K}$, its encoder $\mathbb{E}_{\mathcal{K}}$ and decoder $\mathbb{D}_{\mathcal{K}}$ are consistent with each other if they invert each other. That is, the following properties hold:*

$$\forall\ k\ l\ r\ l', valid\_input_{\mathcal{K}}(l) \Longrightarrow \mathbb{E}_{\mathcal{K}}(k,l) = \lfloor r \rfloor \Longrightarrow \mathbb{D}_{\mathcal{K}}(r{+}{+}l') = \lfloor (k,l,l') \rfloor.$$
$$\forall\ k\ l\ r\ l', \mathbb{D}_{\mathcal{K}}(r{+}{+}l') = \lfloor (k,l,l') \rfloor \Longrightarrow \mathbb{E}_{\mathcal{K}}(k,l) = \lfloor r \rfloor.$$

```
Definition encode_addrmode instance input :=
  match instance with
  ...
  | addr_sib arg1 arg2 arg3 ⇒
    (* Encode ModRM *)
    let ModRM := input in
    let tmp := write_mod ModRM b["00"] in
    let tmp := write_rm tmp b["100"] in
    let result0 := tmp in
    (* Encode SIB *)
    let SIB := zeros 8 in
    let tmp := write_scale SIB arg1 in
    let tmp := write_index tmp arg2 in
    let tmp := write_base tmp arg3 in
    let index := read_index tmp in
    let base := read_base tmp in
    do _ ← assert(index ≠ b["100"]);
    do _ ← assert(base ≠ b["101"]);
    let result1 := tmp in
    (* Concatenate the results of
       encoding ModRM and SIB *)
    Some (result0++result1)
  | ...
  end.
```

```
Definition decode_addrmode bs :=
  ...
  if BF_addr_sib bs then
    (* Revert the encoding of ModRM *)
    let ori := clear_mod bs in
    let ori := clear_rm ori in
    let ori1 := ori in
    do remains ← skipn bs 8; (* Skip ModRM *)
    (* Decode SIB to get the arguments
       and revert the encoding of SIB *)
    let bs := remains in
    let arg3 := read_base bs in
    let ori := clear_base bs in
    let arg2 := read_index ori in
    let ori := clear_index ori in
    let arg1 := read_scale ori in
    let ori := clear_scale ori in
    let ori2 := ori in
    do remains ← skipn bs 8;  (* Skip SIB *)
    (* Return the result *)
    Some(addr_sib arg1 arg2 arg3,
         ori1++ori2, remains)
  else if BF_addr_r bs then ...
    ...
```

```
Definition encode_instr instance input :=
  match instance with
  |AddGvEv arg1 arg2 ⇒
    ...
    let tmp := write_reg_op ModRM arg1 in
    do tmp ← encode_addrmode arg2 tmp;
    ...
  | ...
  end.

Definition decode_instr bs :=
  if BF_AddGvEv bs then
    ...
    do arg2, ori, remains ←
       decode_addrmode bs;
    let arg1 := read_reg_op ori in
    let ori := clear_reg_op ori in
    ...
```

```
Definition BF_addr_sib bs :=
  let ModRM := firstn bs 8 in
  (* mod = 0b00 ∧ rm = 0b100 *)
  let result0 :=
    (ModRM & b["11000111"]) = b["00000100"] in
  let tmp := skipn bs 8 in
  let SIB := firstn tmp 8 in
  (* index ≠ 0b100 *)
  let result10 :=
    (SIB & b["00111000"]) ≠ b["00100000"] in
  (* base ≠ 0b101 *)
  let result11 :=
    (SIB & b["00000111"]) ≠ b["00000101"] in
  result0 ∧ result10 ∧ result11.

Definition BF_AddGvEv bs :=
  let Opcode := firstn bs 8 in
  (Opcode & b["11111111"]) = b["00000011"].
```

**Fig. 9.** Encoders and decoders generated from the running example

We first discuss how the proof for the first property in Theorem 1 is generated. Here, the assumption $valid\_input_{\mathcal{K}}(l)$ asserts that all the bits in $l$ that may be modified by $\mathbb{E}_{\mathcal{K}}$ must be 0. This is necessary to ensure that the decoder can revert the resulting bit string back to its initial state by setting them to 0 (i.e., the second result of decoding is the same as $l$).

The proof proceeds by induction on the structure of $k$. For each branch $\mathcal{B}$ with the pattern $\mathcal{P}$, we generate a lemma and its proof that the decision procedure generated from $[\![\mathcal{P}]\!]_{cond}$ as described in Sect. 5.2 always returns true given any bit string generated by the encoder for $\mathcal{P}$. With this lemma, the proof for the "symmetric" case where the decoder takes the same branch as the encoder reduces to proving that the encoder and decoder generated from $\mathcal{P}$ are inverses of each other. This proof is straightforward by the definitions of $\mathcal{G}_{\mathbb{E}}$ and $\mathcal{G}_{\mathbb{D}}$

in Sect. 5. An important point to note is that, for any pattern `cls` $\%i$, we need to recursively apply the consistency lemma for its corresponding class, which in turn requires us to establish a *valid_input* assumption. By the disjointness property in Sect. 4.4, we can easily conclude that the encoding of sub-components does not interfere with each other, thereby the desired *valid_input* assumption can be derived.

To finish the proof, we need to show that the "asymmetric" cases are not possible. For each asymmetric branch $\mathcal{B}'$ with the pattern $\mathcal{P}'$, we have that $[\![\mathcal{P}']\!]_{cond}$ holds by the decision procedure guarding this branch. Furthermore, by the above reasoning, $[\![\mathcal{P}]\!]_{cond}$ holds. We hence have that the conjunction of $[\![\mathcal{P}]\!]_{cond}$ and $[\![\mathcal{P}']\!]_{cond}$ holds. However, this contradicts with the uniqueness property given in Sect. 4.4. Therefore, the decoder can never go into a branch different from the encoder. Continue with our running example, suppose we are proving the consistency of the encoder and decoder for *Addrmode*. Further suppose we are working on the branch with the constructor *addr_sib*. Then, the verification condition for the asymmetric case with the constructor *addr_r* is

$$\forall bs, (read_{mod}\ bs = \texttt{0b00} \wedge read_{rm}\ bs = \texttt{0b100}\ldots) \wedge (read_{mod}\ bs = \texttt{0b11})$$

which cannot possibly hold (for simplicity we omit the conditions for *index* and *base*). We note that such condition can be easily checked by any SMT solver with the theory of bit-vectors, and we use Z3 [5] to validate them. This checking can also be directly formalized in Coq, which we plan to do in the future.

Finally, the second property in Theorem 1 can be proved by induction on $k$ in a similar fashion. We elide a discussion of its proof.

## 6.2 Synthesizing the Proof of Soundness

As we have discussed in Sect. 4.3, the relational specifications extracted from CSLED specifications are tightly related to the actual instruction formats. Thus, it is reasonable to check the soundness of the generated encoders and decoders against these specifications. The relational specifications are easily translated into Coq definitions and we shall use the same notations. The soundness of encoders and decoders is then stated as follows:

**Theorem 2 (Soundness of Encoders and Decoders).** *Given any class $\mathcal{K}$, its encoder $\mathbb{E}_{\mathcal{K}}$ is sound if the following property holds:*

$$\forall\ k\ l\ r\ l', \mathbb{E}_{\mathcal{K}}(k, l) = \lfloor r \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ r.$$

*Similarly, its encoder $\mathbb{D}_{\mathcal{K}}$ is sound if the following holds:*

$$\forall\ k\ l\ r\ l', \mathbb{D}_{\mathcal{K}}(r{+}{+}l') = \lfloor (k, l, l') \rfloor \Longrightarrow \mathbb{R}[\![\mathcal{K}]\!]\ k\ r.$$

The soundness of encoder is easily proved by induction on the structure of $k$. We need to exploit the well-formedness conditions of CSLED specifications as for the consistency proofs at relevant points. The soundness of decoder is a corollary of the soundness of encoder and the second consistency property.

## 7 Evaluation

Besides the CSLED language, our framework has two major parts: *1)* the algorithms for generating encoders, decoders and their proofs and *2)* a Coq library containing the definitions and properties of basic types (including bits, bytes and bit strings) and a collection of automation tactics (Ltac definitions) for proof synthesis. The generation algorithms amount to 5,193 lines of C++ code (excluding comments and empty lines, and likewise for the following statistics). The Coq library amounts to 1,036 lines of Coq code (written in Coq 8.11.0 and counted using `coqwc`). We also make use of the monad definitions and some basic data formats in CompCert's library [13]. The whole framework took six person months to develop.

**Table 2.** The lines of generated Coq code

| Component | Lines of definitions | Lines of proofs |
|---|---:|---:|
| Relational specification | 1762 | 0 |
| AST, encoder and decoder | 5677 | 0 |
| Verification conditions | 37011 | 4402 |
| Consistency proof | 295 | 30841 |
| Soundness proof | 60 | 7193 |
| Total | 44805 | 42436 |

To evaluate the effectiveness of our framework, we have written a CSLED specification for a total of 186 representative X86-32 instructions which cover the operands with the most complicated formats (e.g., addressing modes) and are sufficient for supporting the assembling process in CompCert's X86-32 backend. The specification is very succinct, containing only 260 lines of CSLED code. From this specification, our framework *automatically* generates around 87k lines of Coq code which form the verified encoder and decoder. The lines of Coq definitions and proofs for individual components are shown in Table 2. Note that the verification conditions account for a major part of the definitions because we need to consider all the possible combinations of structural conditions for the proofs of consistency and soundness. The Coq proofs related to verification conditions are for identifying the concrete forms of structural conditions. As expected, the consistency proof is the most complicated one among all the proofs.

To evaluate the performance of the generated encoder and decoder, we randomly generate four sets of instructions, encode them into bit strings, and decode the bit strings back. The executable encoder and decoder are obtained by extracting Coq definitions into OCaml programs and compiling with OCaml 4.08.0. We repeat this experiment for 30 times on a machine with Intel(R) i7-4980HQ CPU@2.8 GHz and 16 GB memory. For comparison, we conduct the same experiments on the hand-written encoder and decoder in the X86-32 back-end of CompCertELF [20]. The results are shown in Table 3. For each test case, it shows the

**Table 3.** Performance evaluation

| No. of Instr. | CSLED | | | | Hand-Written | | | |
|---|---|---|---|---|---|---|---|---|
| | Enc. Time (s) | | Dec. Time (s) | | Enc. Time (s) | | Dec. Time (s) | |
| | Med | Var.(%) | Med | Var.(%) | Med | Var.(%) | Med | Var.(%) |
| 6000 | 0.32 | 0.00 | 0.56 | 0.00 | 0.01 | 0.00 | 0.01 | 0.00 |
| 12000 | 0.64 | 0.00 | 1.12 | 0.00 | 0.01 | 0.00 | 0.02 | 0.00 |
| 18000 | 0.98 | 0.03 | 1.70 | 0.15 | 0.02 | 0.00 | 0.03 | 0.01 |
| 60000 | 3.11 | 0.16 | 5.43 | 0.01 | 0.08 | 0.00 | 0.09 | 0.01 |

numbers of randomly generated instructions and the median time (in seconds) and the variance (in percentage) for encoding and decoding. We observe that the automatically generated encoder and decoder perform reasonably well, but significantly slower than the hand-written ones. This is because 1) the hand-written encoder and decoder in CompCertELF currently supports significantly less instructions (about 20) than the CLSED ones due to the complexity in manual implementation, and 2) the hand-written ones are manually optimized while the auto-generated ones are not optimized at all. We plan to solve the above issues by optimizing our generation algorithms in the future.

## 8 Related Work and Conclusion

We compare our framework with existing work on specification languages of instruction sets, verified parsing and pretty printing, and formalized ISAs.

There exists a lot of work on developing languages for specifying ISAs. Their major deficiency is the lack of formal guarantees. For example, the nML specification language employs attribute grammars to describe instruction sets [7]. For another example, EEL uses machine independent primitives to provide syntactic and semantic information of instructions [12]. The most relevant work in this category is the SLED language which our CSLED is based upon [15]. The patterns in SLED can only describe constraints on tokens and fields. By contrast, CSLED contains class patterns for accurately characterizing the structures of components. This extension enables CSLED to capture the bijection between the abstract and concrete forms of instructions.

Instruction decoding and encoding are special cases of parsing and pretty printing, respectively. Although there was early work on verifying that parsing and pretty-printing are inverses of each other by formulating them as bijections [1,10], this requirement was perceived as too strong [16]. Most of the recent work on verified parsing and pretty printing are dedicated to verify parser generators based on context-free grammars, regular expressions, parser combinators, or general data formats [3,11,17]. Some of them are also specialized work on verifying the encoder-decoder pairs [6,14,19,21]. They mostly deal with general and ambiguous grammars or specifications where bijection is difficult (if not impossible) to establish. By contrast, we intentionally restrict the expressiveness of CSLED specifications to make proving consistency possible. Specifically, the syntax presented in Fig. 4 implies that CSLED specifications can only match sequences of tokens with finite lengths and shapes, making it strictly weaker than regular expressions, yet sufficiently strong for precisely capture the common instruction formats.

There is also abundant work on the development of formal ISA specifications (e.g., [2,4,8,9]). However, almost all of them focus on the problem of rigorously defining the *semantics* of ISAs (such as their sequential behaviors, concurrency models and interrupt behaviors). Although formalized encoders or decoders (or both) are sometimes generated (e.g., in Coq or Isabelle/HOL), there is no formal verification of the soundness or consistency of instruction encoding and decoding which only concerns the *syntax* of instructions.

In this paper, we have presented a framework for specifying instruction formats and for automatically generating and verifying encoders and decoders based on such specifications. The verified encoders and decoders are consistent with each other (being inverses of each other) and sound (conforming to high-level specifications). Consistency is provable in our framework because our specifications capture the bijections inherently embedded in instruction formats. In the future, we would like to apply this framework to a major part of X86-32 and X86-64 instructions and also to other ISAs, thereby to demonstrate the versatility and scalability of our framework.

**Acknowledgments.** We thank Zhong Shao for his comments and suggestions on this project when it was at an early stage. We are also indebted to anonymous reviewers for their detailed comments. This work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62002217.

## References

1. Alimarine, A., Smetsers, S., Weelden, A., Eekelen, M., Plasmeijer, R.: There and back again: arrows for invertible programming. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell, pp. 86–97. ACM (2005). https://doi.org/10.1145/1088348.1088357
2. Armstrong, A., et al.: ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. Proc. ACM Program. Lang. **3**(POPL), 71:1–71:31 (2019). https://doi.org/10.1145/3290384
3. Barthwal, A., Norrish, M.: Verified, executable parsing. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 160–174. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_12
4. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86–64 user-level instruction set architecture. In: PLDI 2019, pp. 1133–1148. ACM (2019). https://doi.org/10.1145/3314221.3314601
5. De Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

6. Delaware, B., Suriyakarn, S., Pit-Claudel, C., Ye, Q., Chlipala, A.: Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. Proc. ACM Program. Lang. **3**(ICFP), 82:1–82:29 (2019). https://doi.org/10.1145/3341686
7. Fauth, A., Van Praet, J., Freericks, M.: Describing instruction set processors using NML. In: Proceedings of the European Design and Test Conference, pp. 503–507. IEEE (1995). https://doi.org/10.1109/EDTC.1995.470354
8. Fox, A.: Directions in ISA specification. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 338–344. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_23
9. Fox, A., Myreen, M.O.: A trustworthy monadic formalization of the armv7 instruction set architecture. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 243–258. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_18
10. Jansson, P., Jeuring, J.: Polytypic compact printing and parsing. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 273–287. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49099-X_18
11. Jourdan, J.H., Pottier, F., Leroy, X.: Validating lr(1) parsers. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 397–416. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_20
12. Larus, J.R., Schnarr, E.: Eel: machine-independent executable editing. In: PLDI 1995, pp. 291–300. ACM (1995). https://doi.org/10.1145/207110.207163
13. Leroy, X.: The CompCert Verified Compiler (2005–2021). https://compcert.org/
14. Ramananandro, T., et al.: Everparse: verified secure zero-copy parsers for authenticated message formats. In: USENIX Security Symposium, pp. 1465–1482. USENIX Association (2019)
15. Ramsey, N., Fernández, M.F.: Specifying representations of machine instructions. ACM Trans. Program. Lang. Syst. **19**(3), 492–524 (1997). https://doi.org/10.1145/256167.256225
16. Rendel, T., Ostermann, K.: Invertible syntax descriptions: Unifying parsing and pretty printing. In: Proceedings of the 3rd ACM Haskell Symposium on Haskell, pp. 1–12. ACM (2010). https://doi.org/10.1145/1863523.1863525
17. Ridge, T.: Simple, functional, sound and complete parsing for all context-free grammars. In: Jouannaud, J.P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 103–118. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_10
18. Tan, Gang, Morrisett, Greg: Bidirectional grammars for machine-code decoding and encoding. J. Autom. Reason. **60**(3), 257–277 (2017). https://doi.org/10.1007/s10817-017-9429-1
19. Van Geest, M., Swierstra, W.: Generic packet descriptions: verified parsing and pretty printing of low-level data. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Type-Driven Development, pp. 30–40. ACM (2017). https://doi.org/10.1145/3122975.3122979
20. Wang, Y., Xu, X., Wilke, P., Shao, Z.: Compcertelf: verified separate compilation of c programs into elf object files. Proc. ACM Program. Lang. **4**(OOPSLA), 197:1–197:28 (2020). https://doi.org/10.1145/3428265
21. Ye, Q., Delaware, B.: A verified protocol buffer compiler. In: CPP 2019, pp. 222–233. ACM (2019). https://doi.org/10.1145/3293880.3294105

# An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation

Juneyoung Lee[1(✉)], Dongjoo Kim[1], Chung-Kil Hur[1], and Nuno P. Lopes[2]

[1] Seoul National University, Seoul, South Korea
juneyoung.lee@sf.snu.ac.kr
[2] Microsoft Research, Cambridge, UK

**Abstract.** Several automatic verification tools have been recently developed to verify subsets of LLVM's optimizations. However, none of these tools has robust support to verify memory optimizations.

In this paper, we present the first SMT encoding of LLVM's memory model that 1) is sufficiently precise to validate all of LLVM's intra-procedural memory optimizations, and 2) enables bounded translation validation of programs with up to hundreds of thousands of lines of code. We implemented our new encoding in Alive2, a bounded translation validation tool, and used it to uncover 21 new bugs in LLVM memory optimizations, 10 of which have been already fixed. We also found several inconsistencies in LLVM IR's official specification document (LangRef) and fixed LLVM's code and the document so they are in agreement.

## 1 Introduction

Ensuring that LLVM is correct is crucial for the safety and reliability of the software ecosystem. There has been significant work towards this goal including, e.g., formally specifying the semantics of the LLVM IR (intermediate representation). This entails describing precisely what each instruction does and how it handles special cases such as integer overflows, division by zero, or dereferencing out-of-bounds pointers [8,24,26,29,47]. There has also been work on automatic verification of classes of optimizations, such as peephole optimizations [25,31], semi-automated proofs [48], translation validation [20,35,42,44], and fuzzing [23,46]. All this work uncovered several hundred bugs in LLVM.

While there has been great success in improving correctness of scalar optimizations, current verification tools only support basic memory optimizations, if any. Since memory operations can take a significant fraction of a program's run time, memory optimizations are very important for performance. The implementation of these optimizations and related pointer analyses tends to be complex, which further justifies the investment in verifying them.

Verifying programs with memory operations is very challenging and it is hard to scale automatic verification tools that handle these. The main issue lies with pointer aliasing: which objects does a given memory operation access? Without any prior information, a verifier must consider that each operation *may* load or store from any live object (global variables and stack/heap allocations). This creates a big case split for the underlying constraint solver to (attempt to) solve.

Since automatic verification of the source code of memory optimizations is out of reach at the moment, we focus on bounded translation validation [30, 40] (BTV) instead. (Bounded) translation validation consists in verifying that an optimization was correct for a particular input program (up to a bounded unrolling of loops) rather than verifying its correctness for all input programs.

In this paper, we present the first SMT encoding of LLVM's memory model [24] that is precise enough to validate all of LLVM's intraprocedural memory optimizations. The design of the encoding was guided by practical insights of the common aliasing cases in BTV to achieve better performance. For example, we observed that in most cases we can cheaply infer whether a pointer aliases with a locally-allocated or a global object (but not both). Therefore, our encoding case-splits itself on this property rather than leaving that to the SMT solver, as we can cheaply resolve the case split for over 95% of the cases.

The second contribution of this paper is a new semantics for heap allocation for the verification of optimizations for real-world C/C++ programs. Although LLVM's memory model has a reasonable semantics for heap allocations [24], we realized it was not suitable for verifying optimizations. In some programming styles, the result of functions such as `malloc` is not checked against `NULL` and the resulting pointer is dereferenced right away. Since `malloc` can return `NULL` in some executions, we could end up proving that some undesirable optimizations were correct since the program triggers undefined behavior in at least one execution. We propose a new semantics for heap allocations in this paper that is better suited for the verification of optimizations.

The third contribution is the identification of approximations to the SMT encoding such that it is still sufficiently precise to verify (and find bugs) in LLVM's memory optimizations. This is possible since for translation validation we only need to be as precise as LLVM's static analyses (e.g., in the encoding of aliasing rules), and therefore we do not need to consider extremely precise analyses nor arbitrary transformations. Compilers have limited reasoning power by construction in order to keep compilation time reasonable.

We implemented our new SMT encoding of LLVM's memory model in Alive2 [30], a bounded translation validation tool for LLVM. We used Alive2 to find and report 21 previously unknown bugs in LLVM memory optimizations, 10 of which have already been fixed.

To summarize, the contributions of this paper are as follows.

1. The first SMT encoding of LLVM's memory model that is precise enough to verify all of LLVM's intraprocedural memory optimizations.
2. A new semantics for heap allocations for the verification of optimizations of real-world C/C++ programs (Sect. 5.1).
3. A set of approximations to the SMT encoding to further improve the performance of verification without introducing false positives or false negatives in practice (Sect. 9).

4. Thorough evaluation of LLVM's memory model against LLVM's implementation, which uncovered deviations from the model (Sect. 10.3).
5. Identification of 21 previously unknown bugs in LLVM. We present a few examples in Sect. 10.1.

## 2 Overview

Consider the functions below in `C`:[1] a source (original) function on the left and a target (optimized) function on the right. According to the semantics of high-level languages, and also of LLVM IR, a pointer received as argument or a callee cannot guess the address of a memory region allocated within a function. That is, pointer `q` is not aliased with `p`, `r`, nor touched by `g(p+1)`. Although the caller of `f` may guess the address of `q` in practice, that behavior is excluded by the language semantics because `p`'s object (*provenance*) cannot be a fresh one like `q`. If `p` happens to alias `q`, accessing such pointer triggers undefined behavior (UB).

```
1  int f(int *p) {
2    int *q = malloc(4);
3    *q = 42;
4    int *r = g(p+1);
5    *r = 37;
6    return *q;
7  }
```

```
1′ int f(int *p) {
2′   // q removed
3′
4′   int *r = g(p+1);
5′   *r = 37;
6′   return 42;
7′ }
```

The provenance rules allow LLVM to forward the stored value in line 3 to line 6, and therefore line 6′ simply returns 42. As the value stored to `*q` is not used anymore and pointer `q` does not escape, LLVM also removes the heap allocation.

Next we show how to verify this example. Note that we do not require the two programs to be aligned; the example is aligned to make it easier to understand.

### 2.1 Verifying the Example Transformation

We start by defining two auxiliary functions that encode the effect of memory operations on the program state. Let state $S = (m, ub)$ be a pair, where $m$ is a memory and $ub$ a boolean that tracks whether the program has already executed UB or not. Let $p$ be the accessed pointer, and $v$ the stored value. The definition of functions $\overline{\textbf{load}}$ and $\overline{\textbf{store}}$ is as follows:

$$\overline{\textbf{load}}\ p\ S ::= (\ \colorbox{lightgray}{$\textbf{load}(p, S.\mathsf{m})$}\ , (S.\mathsf{m},\ S.\mathsf{ub} \vee \neg\ \colorbox{lightgray}{$\textbf{deref}(p, \mathtt{sizeof}(*p), S.\mathsf{m})$}\ ))$$
$$\overline{\textbf{store}}\ p\ v\ S ::= (\ \colorbox{lightgray}{$\textbf{store}(p, v, S.\mathsf{m})$}\ , S.\mathsf{ub} \vee \neg\ \colorbox{lightgray}{$\textbf{deref}(p, \mathtt{sizeof}(*p), S.\mathsf{m})$}\ )$$

$\overline{\textbf{load}}$ returns a pair with the loaded value and the updated state, where $ub$ is further constrained to ensure that pointer $p$ is dereferenceable for at least the size of the loaded type. Similarly, $\overline{\textbf{store}}$ returns the updated state. The gray boxes ( $\colorbox{lightgray}{$\cdots$}$ ) encode SMT expressions; we describe these in the next section.

[1] We use the syntax of `C` for many of the examples in this paper to make them easier to read, even though we consider the semantics of LLVM IR.

**Table 1.** States and axioms after executing each of the lines of `f`.

| # | **Inputs:** $p, m_0, ub_0$ | # | **Inputs:** $p', m'_0, ub'_0$ |
|---|---|---|---|
| 2 | $S_1 := (m_0, ub_0)$ $A_1 :=$ $q$ is fresh | $2'$ | - |
| 3 | $S_2 := \overline{\textbf{store}}\ q\ 42\ S_1$ | $3'$ | - |
| 4 | $S_3 := (m_\mathrm{g}, S_2.\mathsf{ub} \vee ub_\mathrm{g})$ <br> $A_2 :=$ $r$ is not aliased with $q$ $\wedge$ $m_\mathrm{g}$ agrees with $S_2.\mathsf{m}$ on $q$ | $4'$ | $S'_1 := (m'_\mathrm{g}, ub'_0 \vee ub'_\mathrm{g})$ |
| 5 | $S_4 := \overline{\textbf{store}}\ r\ 37\ S_3$ | $5'$ | $S'_2 := \overline{\textbf{store}}\ r'\ 37\ S'_1$ |
| 6 | $\mathrm{O} := \overline{\textbf{load}}\ q\ S_4$ | $6'$ | $\mathrm{O}' := (42, S'_2)$ |

*1. Encoding the output states.* Table 1 shows the state after executing each of the programs' lines. $p$, $m_0$, and $ub_0$ are SMT variables for the input pointer, and function `f` caller's memory and UB flag, respectively. The target's corresponding variables are primed. Meta variables are upper-cased and SMT variables are lower-cased.

On line 2, $q$ is assigned a pointer to a new object (encoded in axiom $A_1$). On line 3, '`*q = 42`' updates the state using $\overline{\textbf{store}}$.

On line 4, the return value, output memory, and UB of `g(p+1)` are represented with fresh variables $r$, $m_\mathrm{g}$, and $ub_\mathrm{g}$, respectively. Axiom $A_2$ encodes the provenance rules: the return value cannot alias with locally non-escaped pointers (`q`) and only the remaining objects are modified. Line $4'$ does not need these axioms because there are no locally-allocated objects in the target function.

Finally, the outputs $O$ and $O'$ are a pair of return value and state.

*2. Relating the source and target's states.* To prove correctness of a transformation, we must first establish refinement between the input states of the source/-target functions. Refinement ($\sqsupseteq$) is used rather than equality because it is allowed for the source's caller to give less defined inputs than the target's.

$$A_\mathrm{in} := p \sqsupseteq p' \wedge m_0 \sqsupseteq m'_0 \wedge (ub'_0 \implies ub_0)$$

The inputs and outputs of function calls are also related using refinement. For any pair of calls in the source and target functions, if the target's inputs refine those of the source, the target's output also refines the source's output. The example only has one function call pair:

$$A_\mathrm{call} := \Big( S_2.\mathsf{m} \sqsupseteq m'_0 \wedge p+1 \sqsupseteq p'+1 \implies m_\mathrm{g} \sqsupseteq m'_\mathrm{g} \wedge r \sqsupseteq r' \wedge (ub'_\mathrm{g} \implies ub_\mathrm{g})\Big)$$

We can now state the correctness theorem for the example transformation. For any input, if the axioms hold, the output of the target must refine that of the source for some internal nondeterminism in the source (e.g., the address of pointer `q`). Output is refined iff $(i)$ the source triggers UB, or $(ii)$ the target triggers no UB, and the target's return value and memory refine those of the source.

$$\forall p, p', m_0, m_0', ub_0, ub_0', m_g, m_g', ub_g, ub_g' .\ \exists q .\ (A_1 \wedge A_2 \wedge A_{in} \wedge A_{call}) \implies O \sqsupseteq O'$$

## 2.2 Efficiently Encoding LLVM's Memory Model and Refinement

We now present our key ideas for efficiently encoding LLVM's memory model and refinement (the gray boxes) in SMT, which is one of our main contributions.

*1. Pointers.* We represent a pointer as a pair $(bid, o)$ of a block id (i.e., its provenance) and an offset within, so that we can easily detect out-of-bound accesses: accessing $(bid, o)$ in memory $m$ triggers UB unless $0 \leq o < m[bid]$.size, from which $\mathbf{deref}((bid, o), sz, m)$ naturally follows.

*2. Bounding the number of blocks.* Our first observation is that we can safely bound the number of memory blocks for *bounded* translation validation since loops are unrolled for a fixed number of iterations. As a result, we can use a (fixed-length) bit-vector to encode block ids.

For the example source function, four blocks are sufficient: three for pointers `p`, `q`, `r` as they may all point to different blocks, and an extra to represent all the other blocks that are not syntactically present but are accessible by function `g`.

For the sake of simplifying the example, we ignore that `p`, `q`, `r` may be **null**. Our model does not make such assumption; we explain later how null is handled.

*3. Aliasing rules.* Several of the aliasing rules are encoded for free as we can distinguish most blocks by construction. First, we use the most significant bit of the block ids to distinguish local (1) from non-local (0) blocks. Second, we assign constant ids whenever possible (e.g., global variables and stack allocations).

For the example source function, (without loss of generality) we set the block ids of `q`, `p` and the extra block to $100_{(2)}$, $000_{(2)}$, and $011_{(2)}$ (in binary format), respectively. However, we cannot fix the block id of `r` and instead give the constraint that it should be either $000_{(2)}$ or $001_{(2)}$ since `r` may alias with `p` but not with `q`. This establishes the alias constraints in $A_1$ and $A_2$ for free.

*4. Memory accesses.* In order to leverage the fact that each pointer may range over a small number of blocks as seen above, we use one SMT array per block (from an offset to a byte) instead of using a single global array (from a pointer to a byte). For the latter, it becomes harder to exploit non-aliasing guarantees since all stores to different blocks are grouped together.

For the example source function, $m_0$ consists of four arrays $m_0^{(100)}$, $m_0^{(000)}$, $m_0^{(001)}$, $m_0^{(011)}$ for the four blocks. Then since $q$'s block id is $100_{(2)}$, $\overline{\mathbf{store}}\ q\ 42\ S_1$ at line 3 only updates the array $m_0^{(100)}$, leaving the others unchanged. Similarly, $\overline{\mathbf{store}}\ r\ 2\ S_3$ at line 5 only updates $m_0^{(000)}$ and $m_0^{(001)}$ using the SMT if-then-else expression on $r$'s block id. Finally, $\overline{\mathbf{load}}\ q\ S_4$ at line 6 reads from the updated array at $100_{(2)}$, thereby easily realizing that the read value is 42.

*5. Refinement.* The value/memory refinement $\sqsupseteq$ is defined based on a mapping between source and target blocks, which we efficiently encode leveraging the alignment information between source and target as much as possible (Sect. 7).

## 3 LLVM's Memory Model

In this section, we give a brief introduction to LLVM's memory model [24]. In this paper we only consider logical pointers (i.e., integer-to-pointer casts are not supported) and a single address space.

*Memory Block.* A memory block is the unit of memory allocation: each stack or global variable has a distinct block, and heap allocation functions like **malloc** create a fresh block each time they are called. Each block is uniquely identified with a non-negative integer (bid), and has associated properties, including size, alignment, whether it can be written to, whether it is alive, allocation type (heap, stack, global), physical address, and value.

*Pointer.* A pointer is defined as a triple $(\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$, where off is an offset within the block bid, and attrs is a set of attributes that constrain dereferenceability and which operations are allowed.

Pointer arithmetic operations (**gep**) only change the offset, with bid and attrs being carried over. Unlike C, an offset is allowed to go out-of-bounds (OOB). Such pointer, however, cannot be dereferenced like in C (triggers undefined behavior—UB), but can be used for pointer comparisons for example.

LLVM supports several pointer attributes. For example, a **readonly** pointer $p$ cannot be used to store data. However, it is possible to use a non-**readonly** pointer $q$ to store data to the same location as $p$ (provided the block is writable). A **nocapture** pointer cannot escape from a function. For example, when a function returns, no global variable may have a **nocapture** pointer stored (otherwise it is UB).

LLVM has three constant pointers. The **null** pointer is defined as $(0, 0, \emptyset)$. Block 0 is defined as zero sized and not alive. The **undef**[2] pointer is defined as $(\beta, \delta, \emptyset)$, with $\beta, \delta$ being fresh variables for each observation of the pointer. There is also a **poison**[3] pointer.

*Instructions.* We consider the following LLVM memory-related instructions:

- Memory access: **load**, **store**
- Memory allocation: **malloc**, **calloc**, **realloc**, **alloca** (stack allocation)
- Lifetime: **start_lifetime** (for stack blocks), **free** (stack/heap deallocation)

[2] In LLVM, **undef** values are arbitrary values of a given type with the additional property that they can yield a different value each time they are observed. **undef** values can be replaced with any value of the same type, except **poison** values.

[3] A **poison** value taints whole expression trees (e.g., **poison** + 1 = **poison**), and branching on it is UB. Similarly, dereferencing a **poison** pointer is UB.

- Pointer-related: **gep** (pointer arithmetic), **icmp** (pointer comparison)
- Library functions: **memcpy**, **memset**, **memcmp**, **strlen**
- Others: **ptrtoint** (pointer-to-integer cast), **call** (function call).

Unsupported memory instructions are: integer-to-pointer casts, and atomic and volatile memory accesses.

## 4 Encoding Memory Blocks and Pointers in SMT

We describe our new encoding of LLVM's memory model in SMT over the next few sections. We use the theories of UFs (uninterpreted functions), BVs (bit-vectors), and arrays with lambdas [7], with first order quantification. Moreover, we consider that the scope of verification is a single function without loops (or where loops have been previously unrolled).

### 4.1 Memory Blocks

Each memory block is assigned a distinct identifier (a bit-vector number). We further split memory blocks into local and non-local. Local blocks are all those that are allocated within the function under consideration, either on the stack or the heap. Non-local blocks are the remaining ones, including global variables, heap/stack allocations in callers and heap allocations in callees (stack allocations in callees are not observable, since they are deallocated when the called function returns, hence there is no need to consider them).

We use the most significant bit (MSB) to encode whether a block is local (1) or non-local (0). This representation allows the null block to have $\mathsf{bid} = 0$ and be non-local. We refer to the short block id, or $\widetilde{\mathsf{bid}}$, to refer to $\mathsf{bid}$ without the MSB. This is used in cases where it has already been established whether the block is local or not. Example with 4-bit block ids:

```
int g;                  // bid(g) = 0001
void f(int *p) {        // bid(p) = 0xyz (with xyz = arbitrary)
  int a[2];             // bid(a) = 1000
  int *q = malloc(4);   // bid(q) = 1001
}
```

The separation of local and non-local block ids is an efficient way to encode the constraint that pointers of these groups cannot alias with each other. In the example above, argument p cannot alias with either a or q.

As we only consider functions without loops, block ids can be statically assigned for each allocation site.

### 4.2 Pointers

A pointer $\mathsf{ptr} = (\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$ is encoded as a single bit-vector consisting in the concatenation of the three elements. The offset is interpreted as a *signed* number (which is why blocks cannot be larger than half of the address space). Each attribute (such as **readonly**) is encoded with a bit. Example with 2-bit block ids and offsets, and a single attribute (we use `.` to visually separate the elements):

```
void f(char readonly *p, char *q) { // p = 0x.ab.1, q = 0y.cd.0
  char *r = p + 2;                  // r = 0x.(ab+2).1
  char *s = q + 3;                  // s = 0y.(cd+3).0
  char *t = malloc(4);              // t = 10.00.0
}
```

Let $\widetilde{\mathsf{off}}$ be a truncated offset where the least significant bits corresponding to the greatest common divisor of the alignment and sizes of all memory operations are removed. For example, if all operations are 4-byte aligned and they access either 4- or 8-byte values, then $\widetilde{\mathsf{off}}$ has less 2 bits than $\mathsf{off}$ (as these are guaranteed to be always zero when accessing the memory).

## 4.3 Block Properties

Each block has seven associated properties: size, alignment, read-only, liveness, allocation type (heap, stack, global), physical address, and value. Block properties are looked up and updated by memory operations. For example, when doing a store, we need to check if the access is within the bounds of the block.

Except for liveness and value, properties are fixed at allocation time. Liveness is encoded with a bit-vector (one bit per block), and value with arrays (indexed on $\widetilde{\mathsf{off}}$). We use a multi-memory encoding, where we have one array per $\mathsf{bid}$.

The encoding of fixed properties differs for local and non-local blocks. For non-local blocks, we use a UF symbol per property, taking $\widetilde{\mathsf{bid}}$ as argument. For local blocks, we cannot use UFs because for the refinement check some of these would have to be quantified (c.f. Sect. 7) and most, if not all, SMT solvers do not support quantification of UF symbols. Therefore, we encode each of the remaining properties of local blocks as an if-then-else (ITE) expression, which is tailored for each use (e.g., each time an operation needs to lookup a local block's size, we build an ITE expression for the given $\widetilde{\mathsf{bid}}$).

Using ITE expressions to encode properties is less concise than using UFs. However, it is not a disaster for two reasons. Firstly, we only need to consider the local blocks that have been allocated beforehand, since the program cannot access blocks allocated afterward. Secondly, pointers are usually not fully arbitrary. Oftentimes we know statically which type of block they refer to, and even what is the block id, given that pointer arithmetic operations do not change the block id. Therefore, the ITE expressions are usually small in practice. Example with 4-bit block ids and offsets of a source program:

```
int g;                        // g = 0001.0000, size_src(001) = 4
void f() {
  char p[2];                  // p = 1000.0000
  char q[3];                  // q = 1001.0000
```

```
    char *r = ... p or q or g ...
    r[2] = 0;
    char t[1];                      // t = 1010.0000
  }
```

The store in this program is only well defined if the size of block pointed by `r` is greater than 2. This is encoded in SMT as follows:

$$\mathsf{ite}(\mathbf{islocal}(r), \mathsf{ite}(\widetilde{\mathbf{bid}}(r) = 0, 2, 3), \mathsf{size}_{src}(\widetilde{\mathbf{bid}}(r))) > 2$$

Function $\mathbf{islocal}(p)$ is encoded with the SMT extract expression to fetch the MSB of the pointer. Similarly, $\widetilde{\mathbf{bid}}(p)$ extracts the relevant bits from a pointer. The expression for local blocks only needs to consider local blocks 0 and 1, since block 2 (`t`) is only allocated afterward. This allows a simple single pass through the code to generate optimized ITE expressions.

**Value.** Value is defined as an array from short offset to byte (described later in Sect. 6.1). For non-local blocks, only those that are constant are initialized with the respective value. The remaining blocks are allowed to take almost any value. The exception is for pointers: non-local blocks cannot initially have local pointers stored, since the calling environment cannot fabricate local pointers.

Local blocks are initialized with **poison** values using a constant array (i.e., an array that yields the same value for all indexes).

## 4.4 Physical Addresses

If a program observes addresses (through, e.g., pointer-to-integer casting), we need additional constraints to ensure that addresses of blocks that overlap in time are disjoint. Since we are doing translation validation, we have two programs with potentially different sets of locally allocated blocks. Therefore, we need to ensure that non-local blocks' addresses are disjoint from those of local blocks of both programs. This makes the disjointness constraints quite complex.

As an optimization, we split the address space in two: local blocks have MSB = 1 and non-locals have MSB = 0. Since the encoding of address disjointness is quadratic in the worst case (cross-product of blocks), halving the number of blocks is significant. This optimization, however, is an under-approximation of the program's behavior (Sect. 9). After investigating LLVM's optimizations, we believe it is highly unlikely this approximation will cause false negatives.

If a program does not observe any pointer's physical address, neither the block's physical address property nor the disjointness axioms are instantiated. However, when dereferencing a pointer, we need to check if the physical address is sufficiently aligned. When physical addresses are not created, we resort to checking alignment of both of the pointer's block and offset. Since in this case physical addresses are not observed (and therefore not constrained by the program using, e.g., pointer comparisons), a block's physical address can take any value, and therefore blocks and offsets must be both sufficiently aligned to ensure that physical pointers are aligned in all program executions. This argument justifies why we can soundly discard physical addresses.

**Table 2.** Comparison of two semantics for pointer comparison.

| | Integer comparison | Non-deterministic |
|---|---|---|
| Fold $\mathsf{p} = \mathsf{q}$ to false if $\mathsf{p}.\text{bid} \neq \mathsf{q}.\text{bid}$ | No | Yes |
| Fold $\mathsf{p} + i = \mathsf{q} + i$ to $\mathsf{p} = \mathsf{q}$ | Yes | No |
| Fold $(int)\mathsf{p} = (int)\mathsf{q}$ to $\mathsf{p} = \mathsf{q}$ | Yes | No |
| Fold $\mathsf{p} < \mathsf{q} \wedge \mathsf{p} \neq \mathsf{q}$ to $\mathsf{p} < \mathsf{q}$ | Yes | No |
| Fold $\mathsf{p} < \mathsf{q} \wedge \mathsf{q} \neq null$ to $\mathsf{p} < \mathsf{q}$ | Yes | Potentially |
| Run-time aliasing checks | Yes | Correct, but not useful |
| Analysis of pointers cast from integers | Harder | Easy |

### 4.5 Pointer Comparison

Given two pointers p and q, if a program learns that q is placed right after p in memory, the program can potentially change the contents of q without the compiler realizing it. Detecting the existence of such code is impossible in general, hence restricting the ways a program can learn the layout of objects in memory is important to make pointer analyses fast yet precise.

A way the memory layout can leak is through pointer comparison. For example, what should $\mathsf{p} < \mathsf{q}$ return if these point to different memory blocks? If it is a well-defined operation (i.e., simply compares their integer values), it leaks memory layout information. An alternative is to return a non-deterministic value to prevent layout leaks, the formal semantics of which is defined at [24].

We found that there are pros and cons of both semantics for the comparison of pointers of different blocks, and that neither of them covers all optimizations that LLVM performs. Table 2 summarizes the effects on each of the optimizations.

We decided to implement the integer comparison semantics, as LLVM performs all the optimizations above and its alias analyses (AA) mostly give up when they encounter an integer-to-pointer cast. In summary, we have to remove the first optimization from LLVM to make it sound. Additionally, we make it harder to improve LLVM's AA algorithms w.r.t. to pointers cast from integers.

### 4.6 Bounding the Maximum Number of Blocks

Since we assume that programs do not have loops, we can statically bound the maximum number of both local and non-local blocks a program may observe.

The maximum number of local blocks in the source and target programs, respectively, $N^{src}_{local}$ and $N^{tgt}_{local}$, is computed by counting the number of heap and stack allocation instructions. Note that this is an upper-bound because not all allocation sites may be reachable in practice.

For non-local blocks, we cannot see their definitions as with local blocks, except for global variables. Nevertheless, we can still bound the maximum number of observed blocks. It is sufficient to count the number of instructions that may return non-local pointers, such as function calls and pointer loads. In addition, we consider a null block when needed (if the null pointer may be observed).

To encode the behavior of source and target programs, we need $N^{src}_{nonlocal} + N^{tgt}_{nonlocal}$ non-local blocks in the worst case, as all referenced pointers may be distinct. However, correct transformations will not have the target program observe more blocks than the source. If the target observes a pointer to a non-local block that was not observed in the source, we can set that pointer to **poison** because its value is not restricted by the source. Therefore, $N^{src}_{nonlocal}$ non-local blocks are sufficient to allow the target to exhibit *an* incorrect behavior.

The bit-width of $\widetilde{\mathsf{bid}}$ is: $w_{\widetilde{bid}} = \lceil \log_2(max(N^{src}_{nonlocal}, max(N^{src}_{local}, N^{tgt}_{local})))\rceil$. When only local or non-local pointers are used, $w_{bid} = w_{\widetilde{bid}}$, as we know statically if the pointer is local or not. Otherwise, $w_{bid} = w_{\widetilde{bid}} + 1$.

## 5 Memory Allocation

In LLVM, memory blocks can be allocated on the stack (**alloca**), in the heap (e.g., **malloc**, **calloc**, etc.), or as global variables. It is surprisingly non-trivial to find a semantics for memory allocations that allows all of LLVM's optimizations, and rejects undesired transformations. For example, we have to support allocation removal and splitting, introduce new stack allocations and new constant global variables, etc. We explore multiple semantics and show their merits and shortcomings in the context of proving correctness of program transformations.

### 5.1 Heap Allocation

Heap allocation is done through functions such as **malloc**, **calloc**, C++'s `new` operator, etc. We describe semantics for **malloc**; remaining functions can be described in terms of it.

First of all, it is important to note that there are two common idioms used in practice by C programmers when doing memory allocation:

```
int *p = malloc(4);        |   int *p = malloc(4);
*p = 0;                    |   if (p) { *p = 0; }
```

In some programs, like the example on the left, **malloc** is assumed to never return **null**, say non-null assumption. This is mainly because the program does not consume too much memory and it is expected that the computer has enough memory/swap space. In other programs like the one on the right, **malloc** is expected to sometimes return **null**, say may-null assumption. Therefore, the program performs null-ness checks.

Since both programming styles are prevalent, we would like optimizations to be correct for both. This is non-trivial, as the two assumptions are conflicting: with the non-null assumption, it is sound to eliminate **null** checks, but not with the may-null assumption. We now explore several possible semantics to find one that works for both programming styles.

*A. Malloc always succeeds.* Based on the non-null assumption, in this semantics we only consider executions where there is enough space for all allocations to succeed. Regardless of whether the target uses more or less memory than the source, all calls to **malloc** yield non-null pointers. Therefore, for example, deleting unused **malloc** calls is allowed.

However, removing **null** checks of **malloc** is also allowed in this semantics. For example, optimizing the right example above into the left one is sound. This transformation, however, is obviously undesirable.

*B. Malloc only succeeds if there is enough free space.* To solve the problem just described, based on the may-null assumption, we can simulate the behavior of dynamic memory allocation and define **malloc** to return a pointer to a newly created block if there is an empty space in memory, and **null** otherwise. This semantics prevents the removal of **null** checks of **malloc** as it may return **null**.

However, this semantics does not explain removal of unused allocations. It aligns both source and target programs' allocations such that any change in the allocation sequence disrupts the program alignment and thus makes verification fail. For example, the following transformation removing unused **malloc** instructions and replacing comparisons of their output with **null** is not supported:

```
int *x = malloc(4);                   // remove x (unused)
if (x != nullptr) { ... }      ⇒      if (true) { ... }
```

In case there were 0 bytes left in memory, `x` would be **null**, but since LLVM assumes that the program cannot observe the state of the allocator it folds the comparison `x != nullptr` to `true` after eliminating the allocation. This optimization would be flagged as incorrect in this semantics.

LLVM assumes very little about the run-time behavior of memory allocators. This is to support, for instance, garbage collectors, where an allocation may fail but if repeated it may succeed because memory was reclaimed in between. This explains why LLVM folds comparisons with **null** of unused memory blocks, and also contradicts the linear view of allocations of this semantics.

*C. Malloc non-deterministically returns null.* This semantics abstracts the behavior of the memory allocator by (1) allowing **malloc** to non-deterministically return **null** even if there is available space, and (2) only considering executions where there is enough space for all allocations to succeed. This semantics prevents the removal of null checks of **malloc**, which fixes the shortcomings of semantics A, and also allows the removal of unused allocations, which fixes those of semantics B. However, this semantics is too weak and therefore allows other undesirable transformations, like the following:

```
p = malloc(4);
*p = 0;              ⇒        exit();
```

For the sake of proving refinement (Sect. 7), we need just one trace triggering UB (i.e., one particular realization of the non-deterministic choices) for a given

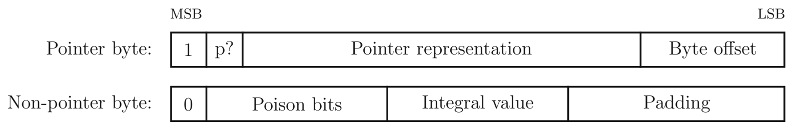


**Fig. 1.** Bit-wise representation of a byte. A pointer byte is poison if 'p?' is zero. A non-pointer byte tracks poison bit-wise.

input to be able to transform the source program into anything for that input. Informally speaking, refinement always picks the worst-case execution for each input. Since the source program executes UB when p is **null**, it is correct to transform the source into any program although that is obviously undesirable.

This semantics is too weak in practice since many programs are written without **null** checks, either assuming the program will not run out of memory, or assuming the program will terminate if it runs out memory. It is not reasonable in practice to allow compilers to break all such programs.

*Our Solution.* As we have seen, there is no single semantics that both allows all desired transformations and rejects undesired ones. While semantics B prevents desired optimizations like allocation removal, semantics A and C allow undesired optimizations, but in a complementary way. For example, removing null checks of **malloc** is allowed in A but not in C. On the other hand, transforming an access of a **malloc**-allocated block without a **null** check beforehand into arbitrary code is allowed in C but not in A.

Therefore, we obtain a good semantics by requiring both A and C: an optimization is correct if it passes the refinement criteria with each of the two semantics. Intuitively, this definition requires the compiler to support the two considered coding styles: semantics A supports the non-null assumption, while semantics C the may-null assumption.

### 5.2 Stack Allocation

The semantics of **alloca**, the stack-allocation instruction, is slightly different from that of **malloc**. LLVM assumes that stack allocations always succeed, since the program will likely crash if there is a stack overflow. That is, **alloca** never returns a **null** pointer.

LLVM performs more optimizations on stack allocations than on heap ones. For example, LLVM can split an allocation into multiple smaller ones or increase the alignment. These transformations can increase memory consumption.

## 6 Encoding Loads and Stores in SMT

We encode the value of memory blocks with several arrays (one per bid): from short offset to byte. We next give the definition of byte and the encoding of memory accessing instructions in SMT.

## 6.1 Byte

There are two types of bytes: *pointer* bytes and *non-pointer* bytes, cf. Fig. 1.

A **pointer byte** has the most significant bit (MSB) set to one. The following bit states whether the byte is poison or not. Next is the pointer representation as described in Sect. 4.2 $(\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$.

Pointers are often longer than one byte, so when storing a pointer to memory we write multiple consecutive bytes. Each of these bytes records the same pointer, but with a different byte offset (the last bits of the byte) to distinguish between the partial bytes of the pointer.

For **non-pointer bytes**, we track whether each of the bits is poison or not. This is not required for pointers, since LLVM does not allow pointer values to be manipulated bit-wise. Non-pointer values can be manipulated bit-wise (e.g., using vectors with element types smaller than 8 bits). Each bit of the integral value is only significant if the corresponding poison bit is zero.

## 6.2 Load and Store Instructions

Load and store instructions are trivially encoded using SMT arrays. These arrays store bytes as described in the previous section. We next describe how LLVM values are encoded to and decoded from our byte representation.

We define two functions, $ty{\Downarrow}(v)$ and $ty{\Uparrow}(b)$, which convert a value $v$ into a byte array and a byte array $b$ back to value, respectively. We show below $ty{\Downarrow}(v)$ when $v \neq$ **poison**. $\mathbf{i}sz$ stands for the integer type with bit-width $sz$. If $sz$ is not a multiple of 8 bits, $v$ is zero-extended first. When $v$ is poison, all poison bits are set to one. $\mathrm{BitVec}(n, b)$ stands for number $n$ with bit-width $b$. Pointer's byte offset is 3 bits because we assume 64-bit pointers.

$$\begin{aligned} \mathbf{i}sz{\Downarrow}(v) \text{ or } \mathbf{float}{\Downarrow}(v) &= \lambda i.\, 0 \mathbin{+\!\!+} 0^8 \mathbin{+\!\!+} \mathrm{bitrepr}(v)[8\times i \ldots 8\times(i+1)-1] \mathbin{+\!\!+} \mathrm{padding} \\ ty{*}{\Downarrow}(v) &= \lambda i.\, 1^2 \mathbin{+\!\!+} \mathrm{bitrepr}(v) \mathbin{+\!\!+} \mathrm{BitVec}(i, 3) \end{aligned}$$

$\mathbf{i}sz{\Uparrow}(b)$ and $\mathbf{float}{\Uparrow}(b)$ return **poison** if any bit is **poison**, or if any of the bytes is a pointer. Otherwise, these functions return the concatenation of the integral values of the bytes.

$ty{*}{\Uparrow}(b)$ returns **poison** if any of the bytes is **poison** or not a pointer, there is more than one distinct pointer value in $b$, or one of the bytes has an incorrect byte offset (they have to be consecutive, from zero to byte size minus one). An exception is reading a non-pointer zero byte, which is interpreted as a null pointer byte. This allows initialization of, e.g., arrays with null pointers with **memset** (which is an idiom commonly used in LLVM IR).

## 6.3 Multi-array Memory

As already described, we use a multi-array encoding for memory, with one array per block id, each indexed on $\widetilde{\mathsf{off}}$. A simpler encoding would have used a single array indexed on $\mathsf{ptr}$. The multi-array encoding is beneficial when we can cheaply compute small aliasing sets for each memory access. In that case, we reduce the

| | | | |
|---|---|---|---|
| Num($sz$) ::= $\{i \mid 0 \le i < 2^{sz}\}$ | BlockID ::= $\mathbb{N}$ | Addr ::= Num(64) | Offset ::= Num(64) |
| PtrAttr ::= {`nocapture`, `readonly`, `readnone`} | Pointer ::= BlockID $\times$ Offset $\times 2^{\text{PtrAttr}}$ | | |
| Value ::= Aggregate $\uplus$ Int $\uplus$ Pointer $\uplus$ Float $\uplus$ {**poison**} | Aggregate ::= list Value | | |
| PtrByte ::= (Pointer $\times \{i \mid 0 \le i < 8\}$) $\uplus$ {**poison**} | NonPtrByte ::= Num(8) $\times$ Num(8) | | |
| Byte ::= PtrByte $\uplus$ NonPtrByte | Bytes ::= Offset $\to$ Byte | Size ::= Num(64) | |
| Align ::= $\{i \mid 0 \le i < 64\}$ | Kind ::= {`stack`, `malloc`, `new`, `global`} | Live ::= bool | |
| Writable ::= bool | MemBlock ::= Addr $\times$ Align $\times$ Kind $\times$ Live $\times$ Writable $\times$ Size $\times$ Bytes | | |
| Memory ::= BlockID $\to$ MemBlock | UB ::= bool | FinalState ::= Value $\times$ Memory $\times$ UB | |

| | | | | |
|---|---|---|---|---|
| $p$ ∈ Pointer | $ag$ ∈ Aggregate | $v$ ∈ Value | $pb$ ∈ PtrByte | $nb$ ∈ NonPtrByte |
| $b$ ∈ Byte | $mb$ ∈ MemBlock | $M$ ∈ Memory | $ub$ ∈ UB | $\mu$ ∈ BlockID $\nrightarrow$ BlockID |

**Fig. 2.** Type definitions and variable naming conventions.

$$\textsc{(value-poison)}\quad \frac{v \in \text{Value}}{\mathbf{poison} \sqsupseteq^{\mu} v} \qquad \textsc{(value-nonptr)}\quad \frac{v \in \text{Int} \uplus \text{Float}}{v \sqsupseteq^{\mu} v} \qquad \textsc{(value-ptr)}\quad \frac{p \sqsupseteq^{\mu}_{\text{ptr}} p'}{p \sqsupseteq^{\mu} p'} \qquad \textsc{(value-aggregate)}\quad \frac{|ag| = |ag'| \qquad \forall i,\, ag[i] \sqsupseteq^{\mu} ag'[i]}{ag \sqsupseteq^{\mu} ag'}$$

$$\textsc{(final-state-ub)}\quad \frac{}{(v, M, \mathbf{true}) \sqsupseteq_{\text{st}} (v', M', ub')} \qquad \textsc{(final-state)}\quad \frac{ub = ub' \qquad \exists \mu,\, v \sqsupseteq^{\mu} v' \wedge M \sqsupseteq^{\mu}_{\text{mem}} M'}{(v, M, ub) \sqsupseteq_{\text{st}} (v', M', ub')}$$

**Fig. 3.** Refinement of value and final state.

case-splitting work on $\mathsf{bid}$ that the SMT solver needs to do, and it enables further formula simplifications like store forwarding.

The multi-array encoding may, however, end up in a larger encoding overall if several of the accesses may alias with too many blocks. For load operations that alias multiple blocks the resulting expression is a linear combination of the loads of each block, e.g., $\mathsf{ite}(\mathsf{bid} = 0, \mathbf{load}(m_0, \widetilde{\mathsf{off}}), \mathsf{ite}(\mathsf{bid} = 1, \mathbf{load}(m_1, \widetilde{\mathsf{off}}), \ldots))$. In this case, it would be more compact to use the single-array encoding. Note that even if we do not know the specific block id, we often know whether a pointer refers to a local or non-local block (e.g., pointers received as argument have unknown block id, but are known to be non-local), and hence splitting the memory in two is usually a good idea (c.f. Sect. 10).

We perform several optimizations that are enabled with this multi-array encoding. We do partial-order reduction (POR) to shrink the potential aliasing of pointers with unknown block id. For example, consider a function with two pointer arguments (`x` and `y`) and one global variable. We assign $\mathsf{bid} = 1$ to the global variable. Then, we estipulate that `x` can only alias blocks with $\mathsf{bid} \le 2$, which is sufficient to access the global variable or another unknown block. Argument `y` is also constrained to only alias blocks with $\mathsf{bid} \le 3$, allowing it to alias with the global variable, the same block as `x`, or a different block. The same is

(pointer)

$$\frac{\begin{array}{c} p.\text{block.live} \Rightarrow p'.\text{block.live} \\ p.\text{offset} = p'.\text{offset} \\ \begin{bmatrix} (\text{isNonLocal}(\{p, p'\}) \wedge p.\text{block.id} = p'.\text{block.id}) \\ \vee\ (\text{isLocal}(\{p, p'\}) \wedge p.\text{block.id} = \mu[p'.\text{block.id}]) \end{bmatrix} \end{array}}{p \sqsupseteq^{\mu}_{\text{ptr}} p'}$$

(memory-map)

$$\frac{\begin{array}{c} \begin{bmatrix} \forall bid,\ \text{isNonLocal}(bid) \\ \Longrightarrow\ M[bid] \sqsupseteq^{\mu}_{\text{blk}} M'[bid] \end{bmatrix} \\ \begin{bmatrix} \forall bid,\ \text{isLocal}(bid) \wedge \mu[bid]\ \text{defined} \\ \Longrightarrow\ M[\mu[bid]] \sqsupseteq^{\mu}_{\text{blk}} M'[bid] \end{bmatrix} \end{array}}{M \sqsupseteq^{\mu}_{\text{mem}} M'}$$

(byte-ptr)

$$\frac{\begin{array}{c} pb.\text{byteoff} = pb'.\text{byteoff} \\ pb.\text{ptr} \sqsupseteq^{\mu}_{\text{ptr}} pb'.\text{ptr} \end{array}}{pb \sqsupseteq^{\mu}_{\text{byte}} pb'}$$

(byte-nonptr)

$$\frac{\begin{array}{c} nb'.\text{p} \mid nb.\text{p} = nb.\text{p} \\ nb.\text{v} \mid nb.\text{p} = nb'.\text{v} \mid nb.\text{p} \end{array}}{nb \sqsupseteq^{\mu}_{\text{byte}} nb'}$$

(byte-zero)

$$\frac{\begin{array}{c} \text{isZeroByte}(b) \\ \text{isZeroByte}(b') \end{array}}{b \sqsupseteq^{\mu}_{\text{byte}} b'}$$

(byte-poison)

$$\frac{\text{isPoisonByte}(b)}{b \sqsupseteq^{\mu}_{\text{byte}} b'}$$

(bytes)

$$\frac{\begin{bmatrix} \forall\, 0 \leq i < mb.\text{size}, \\ mb.\text{bytes}[i] \sqsupseteq^{\mu}_{\text{byte}} mb'.\text{bytes}[i] \end{bmatrix}}{mb \sqsupseteq^{\mu}_{\text{bytes}} mb'}$$

(block)

$$\frac{\begin{array}{cc} mb.\text{live} \Rightarrow mb'.\text{live} & mb.\text{size} = mb'.\text{size} \\ mb.\text{kind} = mb'.\text{kind} & mb.\text{writable} = mb'.\text{writable} \\ mb.\text{align} \leq mb'.\text{align} & mb.\text{live} \Rightarrow mb \sqsupseteq^{\mu}_{\text{bytes}} mb' \end{array}}{mb \sqsupseteq^{\mu}_{\text{blk}} mb'}$$

**Fig. 4.** Refinement of memory and pointers.

done for function calls that return pointers. This POR technique greatly reduces the potential aliasing of unknown pointers without losing precision.

## 7 Verifying Correctness of Optimizations

To verify correctness of LLVM optimizations, we establish a refinement relation between source (or original) and target (or optimized) functions. Equivalence is not used due to undefined behavior and nondeterminism. Compilers are allowed to reduce the set of possible behaviors from the source.

Given functions $f_{src}$ and $f_{tgt}$, set of input and output variables $I_{src}/I_{tgt}$ and $O$ (which include, e.g., memory and the return value), and set of non-determinism variables $N_{src}/N_{tgt}$, $f_{src}$ is refined by $f_{tgt}$ iff:

$$\begin{aligned} \forall I_{src}, I_{tgt}, O_{tgt}\,.\ & \text{valid}(I_{src}, I_{tgt}) \ \wedge\ I_{src} \sqsupseteq I_{tgt} \ \wedge\ \exists N_{src}\,.\,\text{pre}_{\text{src}}(I_{src}, N_{src}) \ \wedge \\ & \big(\exists N_{tgt}\,.\,\text{pre}_{\text{tgt}}(I_{tgt}, N_{tgt}) \ \wedge\ [\![ f_{tgt} ]\!](I_{tgt}, N_{tgt}) = O_{tgt}\big) \\ \Longrightarrow\ & \big(\exists N_{src}\,.\,\text{pre}_{\text{src}}(I_{src}, N_{src}) \ \wedge\ [\![ f_{src} ]\!](I_{src}, N_{src}) \sqsupseteq_{\text{st}} O_{tgt}\big) \end{aligned}$$

Predicate valid$(I_{src}, I_{tgt})$ encodes the global precondition of the input memory and arguments such as disjointness of non-local blocks. Function's preconditions, $\text{pre}_{\text{src}}$ and $\text{pre}_{\text{tgt}}$, include the constraint for disjointness of local blocks. The existential $\text{pre}_{\text{src}}$ constrains the input such that the source function has at least one possible execution. $\sqsupseteq_{\text{st}}$ is the refinement between final states.

Figure 2 shows the definition of final program state which is a tuple of return value, return memory, and UB. A memory is a function from block id to a memory block. A memory block has seven attributes that are described in Sect. 4.3.

Figure 3 shows the definition of refinement of value and final state. For pointers, we cannot simply use equality because local pointers in source and target are internal to each of the functions. Even if they have the same block identifier, they may refer to different allocation sites in the functions (VALUE-PTR). Similarly, the refinement of the final state should consider this difference between local pointers. To address this, we track a mapping $\mu$ between escaped local blocks of the two functions (described next).

### 7.1 Refinement of Memory

Checking refinement of non-local memory blocks is simple as blocks are the same in the source and target functions (e.g., global variables have the same ids in the two functions). Therefore, one just needs to compare blocks of source and target functions with the same id pairwise.

Checking refinement of local blocks is harder but needed when, e.g., the function returns a locally-allocated heap block. This is legal, but block ids in the two functions may not be equal as allocations may have happened in a different order. Therefore, we cannot simply compare local blocks with the same ids.

To check refinement of local blocks, we need to align the two functions' allocations, i.e., we need to find a correspondence between local blocks of the two functions. We introduce a mapping $\mu \in \text{BlockID} \nrightarrow \text{BlockID}$ between target and source local block ids.

Local blocks become related on function calls and return statements, which is when local pointers may be observed. For example, if a function is called with a pointer to a local block as the first argument, $\mu$ should relate that pointer with the first argument of an equivalent function call in the target function.

Figure 4 gives the definition of memory refinement, $M \sqsupseteq^{\mu}_{\text{mem}} M'$, as well as other related relations between memory blocks and pointers. The first rule POINTER describes refinement between source pointer $p$ and target pointer $p'$ with respect to $\mu$. The following four rules define refinement between bytes $b$ and $b'$. In rule BYTE-NONPTR, '$a \,|\, b$' is the bitwise OR operation, and it is used to check the equality of only those bits that are not **poison**. Predicate isZeroByte($b$) holds if $b$ is a **null** pointer or if it is a zero-valued non-pointer byte. This is needed because stores of **null** pointers can be optimized to **memset** instructions.

Rules BYTES and BLOCK define refinement between memory blocks' values and memory blocks, respectively. Rule MEMORY-MAP describes memory refinement with respect to local block mapping $\mu$. $M[bid]$ stands for the memory block with block id $bid$.

The well-formedness of $\mu$ is established in the refinement rules for function calls and return statements. We show these for function calls in the next section. We note that there might be multiple well-formed $\mu$ due to non-determinism.

$$\text{(NONPTR-ARG)}\ \frac{\begin{bmatrix} v, v' \notin \\ \text{Pointer}\end{bmatrix} \quad v \sqsupseteq^{\mu} v'}{v \sqsupseteq_{\text{arg}}^{\mu,sz} v'} \qquad \text{(PTR-ARG-MAPPED)}\ \frac{p \sqsupseteq_{\text{ptr}}^{\mu} p'}{p \sqsupseteq_{\text{arg}}^{\mu,sz} p'} \qquad \text{(PTR-ARG-UNMAPPED)}\ \frac{\begin{array}{c}\text{isLocal}(\{p, p'\}) \\ p.\text{offset} = p'.\text{offset} \\ M[p.\text{bid}] \sqsupseteq_{\text{blk}}^{\mu} M'[p'.\text{bid}]\end{array}}{p \sqsupseteq_{\text{arg}}^{\mu,sz} p'}$$

$$\text{(PTR-ARG-BYVAL)}\ \frac{\begin{array}{c} sz > 0 \quad o = p.\text{offset} \quad o' = p'.\text{offset} \\ mb = M[p.\text{bid}] \quad mb' = M'[p'.\text{bid}] \\ \begin{bmatrix} \forall 0 \le i < sz, \\ mb.\text{bytes}[o+i] \sqsupseteq_{\text{byte}}^{\mu} mb'.\text{bytes}[o'+i] \end{bmatrix}\end{array}}{p \sqsupseteq_{\text{arg}}^{\mu,sz} p'}$$

**Fig. 5.** Refinement between function arguments.

## 8 Function Calls

A call to an unknown function may change the memory arbitrarily (except for, e.g., constant variables and non-escaped local blocks). The outputs in the source and target are, however, related: if the target's inputs refine those of the source, refinement holds between their outputs as well. Alive2 already supported function calls; this section shows how it was extended to support memory.

Let $(M_{in}, v_{in})$ and $(M_{out}, v_{out})$ be the input and output of a function call in the source, and their primed versions, $(M'_{in}, v'_{in})$ and $(M'_{out}, v'_{out})$, those of a function call in the target. Let $\mu_{in}$ be a local block mapping before executing the calls. To state that the outputs are refined if the inputs are refined, we add the following formula to the target's precondition:

$$\Big(M_{in} \sqsupseteq_{\text{mem}}^{\mu_{in}} M'_{in} \wedge \forall i\,.\; v_{in}[i] \sqsupseteq_{\text{arg}}^{\mu_{in}, sz[i]} v'_{in}[i]\Big) \implies \big(M_{out} \sqsupseteq_{\text{mem}}^{\mu_{out}} M'_{out} \wedge v_{out} \sqsupseteq^{\mu_{out}} v'_{out}\big)$$

A call to a function with a pointer to a local block as argument escapes this block, as the callee may, e.g., store that pointer to a global variable. Moreover, any pointer stored in this block also escapes as the callee may traverse the block and grab any pointer stored there, and do so transitively. The updated mapping $\mu_{out} = \text{extend}(\mu_{in}, M_{in}, M'_{in}, v_{in}, v'_{in})$ returns $\mu_{in}$ updated with the relationship between the newly escaped blocks in source and target functions.

Figure 5 shows the definition of refinement between function call arguments in source and target programs. The first rule relates non-pointer arguments. The second one handles pointers that have escaped before these calls. The third rule handles local pointers of blocks that did not escape before these calls, and therefore we need to check if the contents of these block are refined.

The fourth refinement rule handles **byval** pointer arguments. These arguments get a freshly allocated block and the contents of the pointer are copied from the pointer's offset onwards.

## 9 Approximating Program Behavior

In order to speedup verification, we approximate programs' behaviors, which can result in false positives and false negatives. We believe none of these approximations has a significant impact for two reasons: (1) we only need to be as precise as

LLVM's static analyses, i.e., we do not need to support arbitrary optimizations, and (2) we do not consider the compiler to be malicious (which may not be true in certain contexts). Moreover, we conducted an extensive evaluation to support these claims, on which we report in the next section.

*Under-Approximations*

1. Physical addresses of local memory blocks have the MSB set to 1, and non-locals set to 0. This is reasonable if we assume the compiler is not malicious and therefore will not exploit our approximation.
2. We do not consider the case where a (portion of a) global variable is initially **undef**, only **poison** or a regular value.
3. Library functions **strlen**, **memcmp**, and **bcmp** are unrolled for a constant number of times. A precondition is added to constrain the input to be smaller than the unroll factor. In the case of **strlen**, the input pointer is often a constant array. We compute the result straight away in this case.

*Over-Approximations.* The set of local blocks that escape (e.g., whose address is stored into a global variable) is computed per function. This may over-approximate the set of escaped pointers at times because, e.g., a pointer may only escape in a particular branch. LLVM also computes the set of escaped pointers per function.

## 10 Evaluation

We implemented our new memory model in Alive2 [30]. The implementation of the memory model consists in about 3.0 KLoC plus an additional 0.4 KLoC for static analyses for optimization.

We run two set of experiments to both validate our implementation and the formal semantics, and to identify bugs in LLVM. First, we did translation validation of LLVM's unit tests (`test/Transforms`) to increase confidence that we match LLVM's behavior in practice. Second, we run five benchmarks: bzip2, gzip, oggenc, ph7, and SQLite3.

Benchmarks were compiled with `-O3`. Moreover, we disabled type-based aliasing because there is no formal model for this feature yet. During compilation, we emitted pairs of IR files before and after each intra-procedural optimization. We discarded syntactically equal pairs as well as pairs without memory operations.

We used a machine with two Intel Xeon E5-2630 v2 CPUs (total of 12 cores). We set Z3's timeout to 1 min and memory limit to 1 GB. Loops were unrolled once. We used LLVM from 11/Dec (`5e31e22`) and Z3 [33] from 16/Dec (`11477f`).

### 10.1 LLVM Unit Tests

LLVM's `Transforms` unit test suite consists in 6,600 tests totaling 36,600 functions. Alive2 takes about 2.5 h (in parallel) to validate these. By running LLVM's unit tests, we found 21 new bugs in memory optimizations.

**Table 3.** Statistics and results for the single-file benchmarks.

| Program | LoC | Pairs | Time (hours) | Correct | Incorrect | TO | OOM | Unsupported pairs |
|---|---|---|---|---|---|---|---|---|
| bzip2 | 5.1k | 2.3k | 1.9 | 316 | 9 | 574 | 175 | 1.2k |
| gzip | 5.3k | 2.6k | 2.0 | 908 | 4 | 922 | 45 | 737 |
| oggenc | 48k | 1.8k | 2.0 | 433 | 5 | 617 | 49 | 701 |
| ph7 | 43k | 5.6k | 3.4 | 1.2K | 23 | 1.5K | 15 | 2.8k |
| sqlite3 | 141k | 12k | 7.5 | 2.2k | 38 | 2.2K | 48 | 7.8k |

We show below an example of a bug we found. This optimization was shrinking the store from 64 to 32 bits, which is incorrect since the last 32 bits were not copied. This happened because of the mismatch in the load/store's sizes.

```
// i32 *x, *y, *z;                       // i32 *x, *y, *z;
i32 *p = (*x < *y ? x : y);   ⇏   i32 r = (*x < *y ? *x : *y);
*(i64*)z = *(i64*)p;                     *z = r;
```

## 10.2 Benchmarks

Table 3 shows the statistics and results for translation validation. The Pairs column indicates the number of source/optimized function pairs considered for validation. We discarded pairs where the two functions were syntactically equal, as the transformation is then trivially correct. The last column indicates the number of skipped pairs because they use features Alive2 does not yet support.

All the 79 incorrect pairs are due to mismatches between LLVM and the formal semantics. Of these, 74 are related with incorrect handling of **undef** and **poison** values, and the remaining 5 are caused by incorrect load type punning optimizations. This shows that our tool has no false positives.

## 10.3 Specification Bugs

While testing our tool, we found a mismatch in the semantics of the **nonnull** attribute between LLVM's documentation and LLVM's code. The documentation specified that passing a null pointer to a **nonnull** argument triggered UB. However, as illustrated below, LLVM adds **nonnull** to a pointer that may be **poison**. This is incorrect because **poison** can be optimized into any value including null.

```
p = gep inbounds q, 1              p = gep inbounds q, 1
f(p)                     ⇒         f(nonnull p) ; UB if p poison
```

We proposed a new semantics to the LLVM developers, where non-conforming pointers would be considered **poison** rather than UB. This was accepted and we have contributed patches to fix the docs and the incorrect optimizations.

### 10.4 Alias Sets

To show that splitting the memory into multiple arrays is beneficial, we gathered statistics of the alias sets in our benchmarks. More than 96% of the dereferenced pointers turned out to be only local or non-local, but not both. This shows that splitting the memory into local and non-local simplifies the memory encoding.

We also counted the number of memory blocks pointers may alias with. Half of the pointers were aliased with just one block. About 80% of the pointers aliased with at most 3 blocks. This is much less than the median number of blocks functions have. The median of the number of memory blocks was $7 \sim 13$ (varying over programs), and only 10% of the functions had fewer than 3 blocks.

## 11 Related Work

*Semantics of LLVM IR.* The official LLVM IR's specification is written in prose [1]. Vellvm [47] and K-LLVM [29] formalized large subsets of the IR in Coq and K, respectively. [26] clarifies the semantics of **undef** and **poison** and proposes a new **freeze** instruction. [24] formalizes various memory instructions of LLVM. [32] presents a C memory model that supports compilation to that LLVM model.

*Translation validation.* [38] presents a translation validation infrastructure for GCC's intermediate language, using a set of arithmetic/aliasing rules for showing equivalence. LLVM-MD [44] and Peggy [42] verify LLVM optimizations by showing equivalence of source and targets with rewrite rules/equality axioms. They suffer, however, from incomplete axioms for aliasing.

In order to simplify the work of translation validation tools, it is possible to extend the compiler to produce hints (witnesses) [18,36,38,41]. One of these tools, Crellvm [20], is formally verified in Coq.

*Verifying programs with memory using SMT solvers.* SMT solvers have been used before to check equivalence of programs with memory [11,14,21,25,31]. [12] give an encoding of some (but not all) aliasing constraints needed to do translation validation of assembly generated by C compilers.

Other memory models encoded in SMT include one for Solidity (Etherium smart contracts) [16], and for separation logic [37,39]. Several verification tools include SAT/SMT-based (partial) memory models for C [2,9,10] and Java [43].

Several automatic software verification tools, often based on CHCs (constrained Horn clauses), support memory programs [6,13]. For example, both SeaHorn and Cascade use a field-sensitive alias analysis to split the memory [15,45].

SLAyer [4] is an automatic tool for analyzing memory safety of a C program using Z3. Smallfoot [3] verifies assertions written in separation logic.

There have been recent advances in speeding up verification of (SMT) array programs [17,22], from which we could likely benefit.

CompCert [27] splits the memory into local (private) and non-local (public) blocks, similarly to what we do, but assumes that allocations never fail [28]. Work on verifying peephole optimizations for CompCert does not support memory [34].

To support integer-to-pointer casts in CompCert, [5] proposes extending integer values to carry block ids as well. In this model, arithmetic on pointer values yields a symbolic expression. [19] makes the pointer-to-integer cast an instruction that assigns a physical address to the block. Neither of these models supports several optimizations performed by LLVM.

## 12 Conclusion

We presented the first SMT encoding of LLVM's memory model that is sufficiently precise to validate all of LLVM's intra-procedural memory optimizations.

Using our new encoding, we found and reported 21 previously unknown bugs in LLVM memory optimizations, 10 of which have already been fixed.

**Acknowledgement.** This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF-2020R1A2C2011947).

## References

1. LLVM language reference manual. https://llvm.org/docs/LangRef.html
2. Ball, T., Bounimova, E., Levin, V., de Moura, L.: Efficient evaluation of pointer predicates with Z3 SMT solver in SLAM2. Technical Report MSR-TR-2010-24, Microsoft Research (2010), https://www.microsoft.com/en-us/research/publication/efficient-evaluation-of-pointer-predicates-with-z3-smt-solver-in-slam2/
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: FMCO (2006). https://doi.org/10.1007/11804192_6
4. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: memory safety for systems-level code. In: CAV (2011). https://doi.org/10.1007/978-3-642-22110-1_15
5. Besson, F., Blazy, S., Wilke, P.: A concrete memory model for CompCert. In: ITP (2015). https://doi.org/10.1007/978-3-319-22102-1_5
6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: SAS (2013). https://doi.org/10.1007/978-3-642-38856-9_8
7. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV (2002). https://doi.org/10.1007/3-540-45657-0_7
8. Chakraborty, S., Vafeiadis, V.: Formalizing the concurrency semantics of an LLVM fragment. In: CGO (2017). https://doi.org/10.1109/CGO.2017.7863732

9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE (2009). https://doi.org/10.1109/ASE.2009.63
11. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: APLAS (2017). https://doi.org/10.1007/978-3-319-71237-6_7
12. Dahiya, M., Bansal, S.: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In: HVC (2017). https://doi.org/10.1007/978-3-319-70389-3_2
13. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012). https://doi.org/10.1145/2254064.2254112
14. Gupta, S., Saxena, A., Mahajan, A., Bansal, S.: Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In: SAT (2018). https://doi.org/10.1007/978-3-319-94144-8_22
15. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of C/C++ programs. In: SAS (2017). https://doi.org/10.1007/978-3-319-66706-5_8
16. Hajdu, Á., Jovanović, D.: SMT-friendly formalization of the solidity memory model. In: ESOP (2020)
17. Ish-Shalom, O., Itzhaky, S., Rinetzky, N., Shoham, S.: Putting the squeeze on array programs: Loop verification via inductive rank reduction. In: VMCAI (2020). https://doi.org/10.1007/978-3-030-39322-9_6
18. Kanade, A., Sanyal, A., Khedker, U.P.: Validation of GCC optimizers through trace generation. SP&E **39**(6), 611–639 (2009). https://doi.org/10.1002/spe.913
19. Kang, J., Hur, C.K., Mansky, W., Garbuzov, D., Zdancewic, S., Vafeiadis, V.: A formal C memory model supporting integer-pointer casts. In: PLDI (2015). https://doi.org/10.1145/2737924.2738005
20. Kang, J., et al.: Crellvm: Verified credible compilation for LLVM. In: PLDI (2018). https://doi.org/10.1145/3192366.3192377
21. Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification of pointer programs by predicate abstraction. Formal Methods Syst. Des. **52**(3), 229–259 (2018). https://doi.org/10.1007/s10703-017-0293-8
22. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: FMCAD (2015). https://doi.org/10.1109/FMCAD.2015.7542257
23. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: PLDI (2014). https://doi.org/10.1145/2594291.2594334
24. Lee, J., Hur, C.K., Jung, R., Liu, Z., Regehr, J., Lopes, N.P.: Reconciling high-level optimizations and low-level code in LLVM. In: Proceedings of the ACM on Programming Languages 2(OOPSLA), November 2018. https://doi.org/10.1145/3276495
25. Lee, J., Hur, C.K., Lopes, N.P.: AliveInLean: a verified LLVM peephole optimization verifier. In: CAV (2019). https://doi.org/10.1007/978-3-030-25543-5_25
26. Lee, J., et al.: Taming undefined behavior in LLVM. In: PLDI (2017). https://doi.org/10.1145/3062341.3062343
27. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
28. d Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model, version 2. Technical Report RR-7987, INRIA, June 2012. http://hal.inria.fr/hal-00703441

29. Li, L., Gunter, E.L.: -LLVM: a relatively complete semantics of LLVM IR. ECOOP (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.7
30. Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: bounded translation validation for LLVM. In: PLDI (2021). https://doi.org/10.1145/3453483.3454030
31. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with Alive. In: PLDI (2015). https://doi.org/10.1145/2737924.2737965
32. Memarian, K., et al.: Exploring C semantics and pointer provenance. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290380
33. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for CompCert. In: PLDI (2016). https://doi.org/10.1145/2908080.2908109
35. Namjoshi, K.S., Tagliabue, G., Zuck, L.D.: A witnessing compiler: a proof of concept. In: RV (2013). https://doi.org/10.1007/978-3-642-40787-1_22
36. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: SAS (2013). https://doi.org/10.1007/978-3-642-38856-9_17
37. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic modulo theories. In: APLAS (2013). https://doi.org/10.1007/978-3-319-03542-0_7
38. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000). https://doi.org/10.1145/349299.349314
39. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: CAV (2013). https://doi.org/10.1007/978-3-642-39799-8_54
40. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS (1998). https://doi.org/10.1007/BFb0054170
41. Rinard, M.C., Marinov, D.: Credible compilation with pointers. In: RTRV (1999)
42. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for LLVM. In: CAV (2011). https://doi.org/10.1007/978-3-642-22110-159
43. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. In: PLDI (2010). https://doi.org/10.1145/1806596.1806635
44. Tristan, J.B., Govereau, P., Morrisett, J.G.: Evaluating value-graph translation validation for LLVM. In: PLDI (2011). https://doi.org/10.1145/1993316.1993533
45. Wang, W., Barrett, C., Wies, T.: Partitioned memory models for program analysis. In: VMCAI (2017). https://doi.org/10.1007/978-3-319-52234-0_29
46. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: PLDI (2011). https://doi.org/10.1145/1993498.1993532
47. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: POPL (2012). https://doi.org/10.1145/2103656.2103709
48. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formal verification of SSA-based optimizations for LLVM. In: PLDI (2013). https://doi.org/10.1145/2491956.2462164

# Automatically Tailoring Abstract Interpretation to Custom Usage Scenarios

Muhammad Numair Mansur[1](✉), Benjamin Mariano[2], Maria Christakis[1], Jorge A. Navas[3], and Valentin Wüstholz[4]

[1] MPI-SWS, Kaiserslautern and Saarbrücken, Germany
{numair,maria}@mpi-sws.org
[2] The University of Texas at Austin, Austin, USA
bmariano@cs.utexas.edu
[3] SRI International, Menlo Park, USA
jorge.navas@sri.com
[4] ConsenSys, Kaiserslautern, Germany
valentin.wustholz@consensys.net

**Abstract.** In recent years, there has been significant progress in the development and industrial adoption of static analyzers, specifically of abstract interpreters. Such analyzers typically provide a large, if not huge, number of configurable options controlling the analysis precision and performance. A major hurdle in integrating them in the software-development life cycle is tuning their options to custom usage scenarios, such as a particular code base or certain resource constraints.

In this paper, we propose a technique that automatically tailors an abstract interpreter to the code under analysis and any given resource constraints. We implement this technique in a framework, tAIlor, which we use to perform an extensive evaluation on real-world benchmarks. Our experiments show that the configurations generated by tAIlor are vastly better than the default analysis options, vary significantly depending on the code under analysis, and most remain tailored to several subsequent code versions.

## 1 Introduction

*Static analysis* inspects code, without running it, in order to prove properties or detect bugs. Typically, static analysis approximates code behavior, for instance, because checking the correctness of most properties is undecidable. *Performance* is another important reason for this approximation. Typically, the closer the approximation is to the actual code behavior, the less efficient and the more *precise* the analysis is, that is, the fewer false positives it reports. For less tight approximations, the analysis tends to become more efficient but less precise.

Recent years have seen tremendous progress in the development and industrial adoption of static analyzers. Notable successes include Facebook's Infer [7,8] and AbsInt's Astrée [5]. Many popular analyzers, such as these, are based on *abstract interpretation* [12], a technique that abstracts the concrete program semantics and reasons about its abstraction. In particular, program states are abstracted as elements of *abstract domains*. Most abstract interpreters offer a wide range of abstract domains that impact the precision and performance of the analysis. For instance, the Intervals domain [11] is typically faster but less precise than Polyhedra [16], which captures linear inequalities among variables.

In addition to the domains, abstract interpreters usually provide a large number of other options, for instance, whether backward analysis should be enabled or how quickly a fixpoint should be reached. In fact, the sheer number of option combinations (over 6M in our experiments) is bound to overwhelm users, especially non-expert ones. To make matters worse, the best option combinations may vary significantly depending on the code under analysis and the resources, such as time or memory, that users are willing to spend.

In light of this, we suspect that most users resort to using the default options that the analysis designer pre-selected for them. However, these are definitely not suitable for all code. Moreover, they do not adjust to different stages of software development, e.g., running the analysis in the editor should be much faster than running it in a continuous integration (CI) pipeline, which in turn should be much faster than running it prior to a major release. The alternative of enabling the (in theory) most precise analysis can be even worse, since in practice it often runs out of time or memory as we show in our experiments. As a result, the widespread adoption of abstract interpreters is severely hindered, which is unfortunate since they constitute an important class of practical analyzers.

**Our Approach.** To address this issue, we present the first technique that automatically tailors a generic abstract interpreter to a custom usage scenario. With the term *custom usage scenario*, we refer to a particular piece of code and specific resource constraints. The key idea behind our technique is to phrase the problem of customizing the abstract-interpretation configuration to a given usage scenario as an optimization problem. Specifically, different configurations are compared using a cost function that penalizes those that prove fewer properties or require more resources. The cost function can guide the configuration search of a wide range of existing optimization algorithms. This problem of tuning abstract interpreters can be seen as an instance of the more general problem of *algorithm configuration* [31]. In the past, algorithm configuration has been used to tune algorithms for solving various hard problems, such as SAT solving [32,33], and more recently, training of machine-learning models [3,18,52].

We implement our technique in an open-source framework called tAIlor[1], which configures a given abstract interpreter for a given usage scenario using a given optimization algorithm. As a result, tAIlor enables the abstract interpreter to prove as many properties as possible within the resource limit without requiring any domain expertise on behalf of the user.

Using tAIlor, we find that tailored configurations vastly outperform the default options pre-selected by the analysis designers. In fact, we show that this is possible even with very simple optimization algorithms. Our experiments

[1] The tool implementation is found at https://github.com/Practical-Formal-Methods/tailor and an installation at https://doi.org/10.5281/zenodo.4719604.

also demonstrate that tailored configurations vary significantly depending on the usage scenario—in other words, there cannot be a single configuration that fits all scenarios. Finally, most of the generated configurations remain tailored to several subsequent code versions, suggesting that re-tuning is only necessary after major code changes.

**Contributions.** We make the following contributions:

1. We present the first technique for automatically tailoring abstract interpreters to custom usage scenarios.
2. We implement our technique in an open-source framework called tAILOR.
3. Using a state-of-the-art abstract interpreter, CRAB [25], with millions of configurations, we show the effectiveness of tAILOR on real-world benchmarks.

## 2 Overview

We now illustrate the workflow and tool architecture of tAILOR and provide examples of its effectiveness.

**Terminology.** In the following, we refer to an abstract domain with all its options (e.g., enabling backward analysis or more precise treatment of arrays etc.) as an *ingredient*.

As discussed earlier, abstract interpreters typically provide a large number of such ingredients. To make matters worse, it is also possible to combine different ingredients into a sequence (which we call a *recipe*) such that more properties are verified than with individual ingredients. For example, a user could configure the abstract interpreter to first use Intervals to verify as many properties as possible and then use Polyhedra to attempt verification of any remaining properties. Of course, the number of possible configurations grows exponentially in the length of the recipe (over 6M in our experiments for recipes up to length 3).

**Workflow.** The high-level architecture of tAILOR is shown in Fig. 1. It takes as input the code to be analyzed (i.e., any program, file, function, or fragment), a user-provided resource limit, and optionally an optimization algorithm. We focus on time as the constrained resource in this paper, but our technique could be easily extended to other resources, such as memory.

The optimization engine relies on a recipe generator to generate a fresh recipe. To assess its quality in terms of precision and performance, the recipe evaluator computes a cost for the recipe. The cost is computed by evaluating how precise and efficient the abstract interpreter is for the given recipe. This cost is used by the optimization engine to keep track of the best recipe so far, i.e., the one that proves the most properties in the least amount of time. tAILOR repeats this process for a given number of iterations to sample multiple recipes and returns the recipe with the lowest cost.

Zooming in on the evaluator, a recipe is processed by invoking the abstract interpreter for each ingredient. After each analysis (i.e., one ingredient), the evaluator collects the new verification results, that is, the verified assertions. All

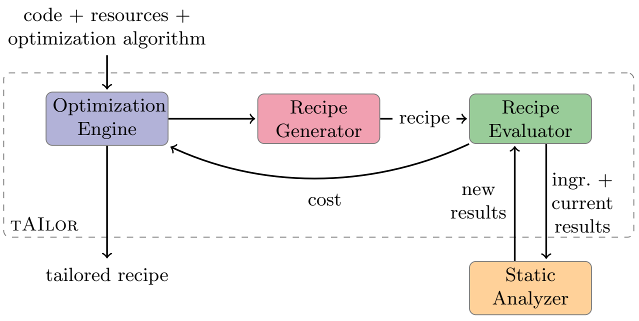


**Fig. 1.** Overview of our framework.

verification results that have been achieved so far are subsequently shared with the analyzer when it is invoked for the next ingredient. Verification results are shared by converting all verified assertions into assumptions. After processing the entire recipe, the evaluator computes a cost for the recipe, which depends on the number of unverified assertions and the total analysis time.

In general, there might be more than one recipe tailored to a particular usage scenario. Naïvely, finding one requires searching the space of all recipes. Section 4.3 discusses several optimization algorithms for performing this search, which tAIlor already incorporates in its optimization engine.

**Examples.** As an example, let us consider the usage scenario where a user runs the Crab abstract interpreter [25] in their editor for instant feedback during code development. This means that the allowed time limit for the analysis is very short, say, 1 s. Now assume that the code under analysis is a program file[2] of the multimedia processing tool ffmpeg, which is used to evaluate the effectiveness of tAIlor in our experiments. In this file, Crab checks 45 assertions for common bugs, i.e., division by zero, integer overflow, buffer overflow, and use after free.

Analysis of this file with the default Crab configuration takes 0.35 s to complete. In this time, Crab proves 17 assertions and emits 28 warnings about the properties that remain unverified. For this usage scenario, tAIlor is able to tune the abstract-interpreter configuration such that the analysis time is 0.57 s and the number of verified properties increases by 29% (i.e., 22 assertions are proved). Note that the tailored configuration uses a completely different abstract domain than the one in the default configuration. As a result, the verification results are significantly better, but the analysis takes slightly longer to complete (although remaining within the specified time limit). In contrast, enabling the most precise analysis in Crab verifies 26 assertions but takes over 6 min to complete, which by far exceeds the time limit imposed by the usage scenario.

While it takes tAIlor 4.5 s to find the above configuration, this is time well invested; the configuration can be re-used for several subsequent code versions. In fact, in our experiments, we show that generated configurations can remain

[2] https://github.com/FFmpeg/FFmpeg/blob/master/libavformat/idcin.c

tailored for at least up to 50 subsequent commits to a file under version control. Given that changes in the editor are typically much more incremental, we expect that no re-tuning would be necessary at all during an editor session. Re-tuning may be beneficial after major changes to the code under analysis and can happen offline, e.g., between editor sessions, or in the worst case overnight.

As another example, consider the usage scenario where Crab is integrated in a CI pipeline. In this scenario, users should be able to spare more time for analysis, say, 5 min. Here, let us assume that the analyzed code is a program file[3] of the curl tool for transferring data by URL, which is also used in our evaluation. The default Crab configuration takes 0.23 s to run and only verifies 2 out of 33 checked assertions. tAIlor is able to find a configuration that takes 7.6 s and proves 8 assertions. In contrast, the most precise configuration does not terminate even after 15 min.

Both scenarios demonstrate that, even when users have more time to spare, the default configuration cannot take advantage of it to improve the verification results. At the same time, the most precise configuration is completely impractical since it does not respect the resource constraints imposed by these scenarios.

## 3 Background: A Generic Abstract Interpreter

Many successful abstract interpreters (e.g., Astrée [5], C Global Surveyor [53], Clousot [17], Crab [25], IKOS [6], Sparrow [46], and Infer [8]) follow the generic architecture in Fig. 2. In this section, we describe its main components to show that our approach should generalize to such analyzers.

**Memory Domain.** Analysis of low-level languages such as C and LLVM-bitcode requires reasoning about pointers. It is, therefore, common to design a *memory domain* [42] that can simultaneously reason about pointer aliasing, memory contents, and numerical relations between them.

*Pointer domains* resolve aliasing between pointers, and *array domains* reason about memory contents. More specifically, array domains can reason about individual memory locations (cells), infer universal properties over multiple cells, or both. Typically, reasoning about individual cells trades performance for precision unless there are very few array elements (e.g., [22,42]). In contrast, reasoning about multiple memory locations (*summarized cells*) trades precision for performance. In our evaluation, we use *Array smashing* domains [5] that abstract different array elements into a single summarized cell. *Logico-numerical domains* infer relationships between program and *synthetic* variables, introduced by the pointer and array domains, e.g., summarized cells.

Next, we introduce domains typically used for proving the absence of runtime errors in low-level languages. *Boolean domains* (e.g., flat Boolean, BDDApron [1]) reason about Boolean variables and expressions. *Non-relational domains* (e.g., Intervals [11], Congruence [23]) do not track relations among different variables, in contrast to *relational domains* (e.g., Equality [35], Zones [41],

[3] https://github.com/curl/curl/blob/master/lib/cookie.c

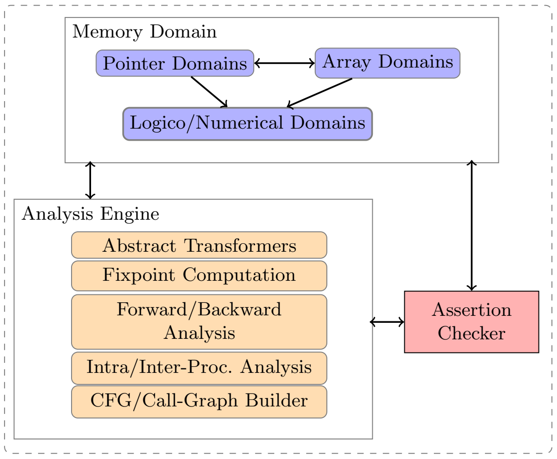


**Fig. 2.** Generic architecture of an abstract interpreter.

Octagons [43], Polyhedra [16]). Due to their increased precision, relational domains are typically less efficient than non-relational ones. *Symbolic domains* (e.g., Congruence closure [9], Symbolic constant [44], Term [21]) abstract complex expressions (e.g., non-linear) and external library calls by uninterpreted functions. *Non-convex domains* express disjunctive invariants. For instance, the DisInt domain [17] extends Intervals to a finite disjunction; it retains the scalability of the Intervals domain by keeping only non-overlapping intervals. On the other hand, the Boxes domain [24] captures arbitrary Boolean combinations of intervals, which can often be expensive.

**Fixpoint Computation.** To ensure termination of the fixpoint computation, Cousot and Cousot introduce *widening* [12,14], which usually incurs a loss of precision. There are three common strategies to reduce this precision loss, which however sacrifice efficiency. First, *delayed widening* [5] performs a number of initial fixpoint-computation iterations in the hope of reaching a fixpoint before resorting to widening. Second, *widening with thresholds* [37,40] limits the number of program expressions (thresholds) that are used when widening. The third strategy consists in applying *narrowing* [12,14] a certain number of times.

**Forward and Backward Analysis.** Classically, abstract interpreters analyze code by propagating abstract states in a *forward* manner. However, abstract interpreters can also perform *backward* analysis to compute the execution states that lead to an assertion violation. Cousot and Cousot [13,15] define a *forward-backward refinement* algorithm in which a forward analysis is followed by a backward analysis until no more refinement is possible. The backward analysis uses invariants computed by the forward analysis, while the forward analysis does not explore states that cannot reach an assertion violation based on the backward analysis. This refinement is more precise than forward analysis alone, but it may also become very expensive.

**Algorithm 1: Optimization engine.**

```
1  Function Optimize($P$, $r_{max}$, $l_{max}$, $i_{dom}$, $i_{set}$, $rec_{init}$, GenerateRecipe,
     Accept) is
2      // Phase 1 (optimize domains)
3      $rec_{best} := rec_{curr} := rec_{init}$
4      $cost_{best} := cost_{curr} :=$ Evaluate($P$, $r_{max}$, $rec_{best}$)
5      for $l := 1$ to $l_{max}$ do
6          for $i := 1$ to $i_{dom} \cdot l$ do
7              $rec_{next} :=$ GenerateRecipe($rec_{curr}$, $l$)
8              $cost_{next} :=$ Evaluate($P$, $r_{max}$, $rec_{next}$)
9              if $cost_{next} < cost_{best}$ then
10                 $rec_{best}, cost_{best} := rec_{next}, cost_{next}$
11             if Accept($cost_{curr}$, $cost_{next}$) then
12                 $rec_{curr}, cost_{curr} := rec_{next}, cost_{next}$
13     // Phase 2 (optimize settings)
14     for $i := 1$ to $i_{set}$ do
15         $rec_{mut} :=$ MutateSettings($rec_{best}$)
16         $cost_{mut} :=$ Evaluate($P$, $r_{max}$, $rec_{mut}$)
17         if $cost_{mut} < cost_{best}$ then
18             $rec_{best}, cost_{best} := rec_{mut}, cost_{mut}$
19     return $rec_{best}$
```

**Intra- and Inter-procedural Analysis.** An *intra-procedural* analysis analyzes a function ignoring the information (i.e., call stack) that flows into it, while an *inter-procedural* analysis considers all flows among functions. The former is much more efficient and easy to parallelize, but the latter is usually more precise.

## 4 Our Technique

This section describes the components of tAIlor in detail; Sects. 4.1, 4.2, 4.3 explain the optimization engine, recipe evaluator, and recipe generator (Fig. 1).

### 4.1 Recipe Optimization

Algorithm 1 implements the optimization engine. In addition to the code $P$ and the resource limit $r_{max}$, it also takes as input the maximum length of the generated recipes $l_{max}$ (i.e., the maximum number of ingredients), a function to generate new recipes GenerateRecipe (i.e., the recipe generator from Fig. 1), and four other parameters, which we explain later.

A tailored recipe is found in two phases. The first phase aims to find the best abstract domain for each ingredient, while the second tunes the remaining analysis settings for each ingredient (e.g., whether backward analysis should be enabled). Parameters $i_{dom}$ and $i_{set}$ control the number of iterations of each phase. Note that we start with a search for the best domains since they have the largest impact on the precision and performance of the analysis.

During the first phase, the algorithm initializes the best recipe $rec_{best}$ with an initial recipe $rec_{init}$ (line 3). The cost of this recipe is evaluated with function Evaluate, which implements the recipe evaluator from Fig. 1. The subsequent nested loop (line 5) samples a number of recipes, starting with the shortest recipes ($l := 1$) and ending with the longest recipes ($l := l_{max}$). The inner loop generates $i_{dom}$ ingredients for each ingredient in the recipe (i.e., $i_{dom} \cdot l$ total iterations) by invoking function GenerateRecipe, and in case a recipe with lower cost is found, it updates the best recipe (lines 9–10). Several optimization algorithms, such as hill climbing and simulated annealing, search for an optimal result by mutating some of the intermediate results. Variable $rec_{curr}$ stores intermediate recipes to be mutated, and function Accept decides when to update it (lines 11–12).

As explained earlier, the purpose of the first phase is to identify the best sequence of abstract domains. The second phase (lines 13–18) focuses on tuning the other settings of the best recipe so far. This is done by randomly mutating the best recipe via MutateSettings (line 15), and updating the best recipe if better settings are found (lines 17–18). After exploring $i_{set}$ random settings, the best recipe is returned to the user (line 19).

### 4.2 Recipe Evaluation

The recipe evaluator from Fig. 1 uses a cost function to determine the quality of a fresh recipe with respect to the precision and performance of the abstract interpreter. This design is motivated by the fact that analysis imprecision and inefficiency are among the top pain points for users [10].

Therefore, the cost function depends on the number of generated warnings $w$ (that is, the number of unverified assertions), the total number of assertions in the code $w_{total}$, the resource consumption $r$ of the analyzer, and the resource limit $r_{max}$ imposed on the analyzer:

$$cost(w, w_{total}, r, r_{max}) = \begin{cases} \dfrac{w + \dfrac{r}{r_{max}}}{w_{total}}, & \text{if } r \leq r_{max} \\ \infty, & \text{otherwise} \end{cases}$$

Note that $w$ and $r$ are measured by invoking the abstract interpreter with the recipe under evaluation. The cost function evaluates to a lower cost for recipes that improve the precision of the abstract interpreter (due to the term $w/w_{total}$). In case of ties, the term $r/r_{max}$ causes the function to evaluate to a lower cost for recipes that result in a more efficient analysis. In other words, for two recipes resulting in equal precision, the one with the smaller resource consumption is assigned a lower cost. When a recipe causes the analyzer to exceed the resource limit, it is assigned infinite cost.

## 4.3 Recipe Generation

In the literature, there is a broad range of optimization algorithms for different application domains. To demonstrate the generality and effectiveness of TAILOR, we instantiate it with four adaptations of three well-known optimization algorithms, namely random sampling [38], hill climbing (with regular restarts) [48], and simulated annealing [36,39]. Here, we describe these algorithms in detail, and in Sect. 5, we evaluate their effectiveness.

Before diving into the details, let us discuss the suitability of different kinds of optimization algorithms for our domain. There are algorithms that leverage mathematical properties of the function to be optimized, e.g., by computing derivatives as in Newton's iterative method. Our cost function, however, is evaluated by running an abstract interpreter, and thus, it is not differentiable or continuous. This constraint makes such analytical algorithms unsuitable. Moreover, evaluating our cost function is expensive, especially for precise abstract domains such as Polyhedra. This makes algorithms that require a large number of samples, such as genetic algorithms, less practical.

Now recall that Algorithm 1 is parametric in how new recipes are generated (with GENERATERECIPE) and accepted for further mutations (with ACCEPT). Instantiations of these functions essentially constitute our search strategy for a tailored recipe. In the following, we discuss four such instantiations. Note that, in theory, the order of recipe ingredients matters. This is because any properties verified by one ingredient are converted into assumptions for the next, and different assumptions may lead to different verification results. Therefore, all our instantiations are able to explore different ingredient orderings.

**Random Sampling.** Random sampling (RS) just generates random recipes of a certain length. Function ACCEPT always returns *false* as each recipe is generated from scratch, and not as a result of any mutations.

**Domain-Aware Random Sampling.** RS might generate recipes containing abstract domains of comparable precision. For instance, the Octagons domain is typically strictly more precise than Intervals. Thus, a recipe consisting of these domains is essentially equivalent to one containing only Octagons.

Now, assume that we have a partially ordered set (poset) of domains that defines their ordering in terms of precision. An example of such a poset for a particular abstract interpreter is shown in Fig. 3. An optimization algorithm can then leverage this information to reduce the search space of possible recipes. Given such a poset, we therefore define domain-aware random sampling (DARS), which randomly samples recipes that do not contain abstract domains of comparable precision. Again, ACCEPT always returns *false*.

**Simulated Annealing.** Simulated annealing (SA) searches for the best recipe by mutating the current recipe $rec_{curr}$ in Algorithm 1. The resulting recipe ($rec_{next}$), if accepted on line 12, becomes the new recipe to be mutated. Algoirthm 2 shows an instantiation of GENERATERECIPE, which mutates a given recipe such that the poset precision constraints are satisfied (i.e., there are no domains of comparable precision). A recipe is mutated either by adding new ingredients with

**Algorithm 2: A recipe-generator instantiation.**

```
1  Function GenerateRecipe($rec$, $l_{max}$) is
2      $act$ := RandomAction({ADD: 0.2, MOD: 0.8})
3      if $act$ = ADD $\wedge$ Len($rec$) $< l_{max}$ then
4          $ingr_{new}$ := RandomPosetLeastIncomparable($rec$)
5          $rec_{mut}$ := AddIngredient($rec$, $ingr_{new}$)
6      else
7          $ingr$ := RandomIngredient($rec$)
8          $act_m$ := RandomAction({GT: 0.5, LT: 0.3, INC: 0.2})
9          if $act_m$ = GT then
10             $ingr_{new}$ := PosetGreaterThan($ingr$)
11         else if $act_m$ = LT then
12             $ingr_{new}$ := PosetLessThan($ingr$)
13         else
14             $rec_{rem}$ := RemoveIngredient($rec$, $ingr$)
15             $ingr_{new}$ := RandomPosetLeastIncomparable($rec_{rem}$)
16         $rec_{mut}$ := ReplaceIngredient($rec$, $ingr$, $ingr_{new}$)
17     if $\neg$PosetCompatible($rec_{mut}$) then
18         $rec_{mut}$ := GenerateRecipe($rec$, $l_{max}$)
19     return $rec_{mut}$
```

20% probability or by modifying existing ones with 80% probability (line 2). The probability of adding ingredients is lower to keep recipes short.

When adding a new ingredient (lines 4–5), Algorithm 2 calls RandomPosetLeastIncomparable, which considers all domains that are incomparable with the domains in the recipe. Given this set, it randomly selects from the domains with the least precision to avoid adding overly expensive domains. When modifying a random ingredient in the recipe (lines 7–16), the algorithm can replace its domain with one of three possibilities: a domain that is immediately more precise (i.e., not transitively) in the poset (via PosetGreaterThan), a domain that is immediately less precise (via PosetLessThan), or an incomparable domain with the least precision (via RandomPosetLeastIncomparable). If the resulting recipe does not satisfy the poset precision constraints, our algorithm retries to mutate the original recipe (lines 17–18).

For simulated annealing, Accept returns *true* if the new cost (for the mutated recipe) is less than the current cost. It also accepts recipes whose cost is higher with a certain probability, which is inversely proportional to the cost increase and the number of explored recipes. That is, recipes with a small cost increase are likely to be accepted, especially at the beginning of the exploration.

**Hill Climbing.** Our instantiation of hill climbing (hc) performs regular restarts. In particular, it starts with a randomly generated recipe that satisfies the poset precision constraints, generates 10 new valid recipes, and restarts with a random recipe. Accept returns *true* only if the new cost is lower than the best cost, which is equivalent to the current cost.

## 5 Experimental Evaluation

To evaluate our technique, we aim to answer the following research questions:

**RQ1:** Is our technique effective in tailoring recipes to different usage scenarios?
**RQ2:** Are the tailored recipes optimal?
**RQ3:** How diverse are the tailored recipes?
**RQ4:** How resilient are the tailored recipes to code changes?

### 5.1 Implementation

We implemented tAIlor by extending Crab [25], a parametric framework for modular construction of abstract interpreters[4]. We extended Crab with the ability to pass verification results between recipe ingredients as well as with the four optimization algorithms discussed in Sect. 4.3.

Table 1 shows all settings and values used in our evaluation. The first three settings refer to the strategies discussed in Sect. 3 for mitigating the precision loss incurred by widening. For the initial recipe, tAIlor uses Intervals and the Crab default values for all other settings (in bold in the table). To make the search more efficient, we selected a representative subset of all possible setting values.

Crab uses a DSA-based [26] pointer analysis and can, optionally, reason about array contents using array smashing. It offers a wide range of logico-numerical domains, shown in Fig. 3. The `bool` domain is the flat Boolean domain, `ric` is a reduced product of Intervals and Congruence, and `term(int)` and `term(disInt)` are instantiations of the Term domain with `intervals` and `disInt`, respectively. Although Crab provides a bottom-up inter-procedural analysis, we use the default intra-procedural analysis; in fact, most analyses deployed in real usage scenarios are intra-procedural due to time constraints [10].

### 5.2 Benchmark Selection

For our evaluation, we systematically selected popular and (at some point) active C projects on GitHub. In particular, we chose the six most starred C repositories

**Table 1.** Crab settings and their possible values as used in our experiments. Default settings are shown in bold.

| **Setting** | **Possible values** |
|---|---|
| `NUM_DELAY_WIDEN` | $\{\mathbf{1}, 2, 4, 8, 16\}$ |
| `NUM_NARROW_ITERATIONS` | $\{1, \mathbf{2}, 3, 4\}$ |
| `NUM_WIDEN_THRESHOLDS` | $\{\mathbf{0}, 10, 20, 30, 40\}$ |
| Backward Analysis | $\{\mathbf{OFF}, \mathit{ON}\}$ |
| Array Smashing | $\{\mathit{OFF}, \mathbf{ON}\}$ |
| Abstract Domains | All domains in Fig. 3 |

[4] Crab is available at https://github.com/seahorn/crab.

**Table 2.** Overview of projects.

| Project | Description |
|---|---|
| CURL | Tool for transferring data by URL |
| DARKNET | Convolutional neural-network framework |
| FFMPEG | Multimedia processing tool |
| GIT | Distributed version-control tool |
| PHP-SRC | PHP interpreter |
| REDIS | Persistent in-memory database |

with over 300 commits that we could successfully build with the Clang-5.0 compiler. We give a short description of each project in Table 2.

For analyzing these projects, we needed to introduce properties to be verified. We, thus, automatically instrumented these projects with four types of assertions that check for common bugs; namely, division by zero, integer overflow, buffer overflow, and use after free. Introducing assertions to check for runtime errors such as these is common practice in program analysis and verification.

As projects consist of different numbers of files, to avoid skewing the results in favor of a particular project, we randomly and uniformly sampled 20 LLVM-bitcode files from each project, for a total of 120. To ensure that each file was neither too trivial nor too difficult for the abstract interpreter, we used the number of assertions as a complexity indicator and only sampled files with at least 20 assertions and at most 100. Additionally, to guarantee all four assertion types were included and avoid skewing the results in favor of a particular assertion type, we required that the sum of assertions for each type was at least 70 across all files—this exact number was largely determined by the benchmarks.

Overall, our benchmark suite of 120 files totals 1346 functions, 5557 assertions (on average 4 assertions per function), and 667927 LLVM instructions (Table 3).

## 5.3 Results

We now present our experimental results for each research question. We performed all experiments on a 32-core Intel ® Xeon ® E5-2667 v2 CPU @ 3.30 GHz machine with 264 GB of memory, running Ubuntu 16.04.1 LTS.

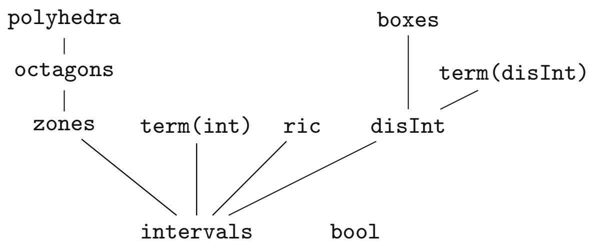


**Fig. 3.** Comparing logico-numerical domains in Crab. A domain $d_1$ is less precise than $d_2$ if there is a path from $d_1$ to $d_2$ going upward, otherwise $d_1$ and $d_2$ are incomparable.

**Table 3.** Benchmark characteristics (20 files per project). The last three columns show the number of functions, assertions, and LLVM instructions in the analyzed files.

| **Project** | **Functions** | **Assertions** | **LLVM instructions** |
|---|---|---|---|
| CURL | 306 | 787 | 50 541 |
| DARKNET | 130 | 958 | 55 847 |
| FFMPEG | 103 | 888 | 27 653 |
| GIT | 218 | 768 | 102 304 |
| PHP-SRC | 268 | 1031 | 305 943 |
| REDIS | 321 | 1125 | 125 639 |
| **Total** | 1346 | 5557 | 667 927 |

**RQ1: Is Our Technique Effective in Tailoring Recipes to Different Usage Scenarios?** We instantiated TAILOR with the four optimization algorithms described in Sect. 4.3: RS, DARS, SA, and HC. We constrained the analysis time to simulate two usage scenarios: 1 s for instant feedback in the editor, and 5 min for feedback in a CI pipeline. We compare TAILOR with the default recipe (DEF), i.e., the default settings in CRAB as defined by its designer after careful tuning on a large set of benchmarks over the years. DEF uses a combination of two domains, namely, the reduced product of Boolean and Zones. The other default settings are in Table 1.

For this experiment, we ran TAILOR with each optimization algorithm on the 120 benchmark files, enabling optimization at the granularity of files. Each algorithm was seeded with the same random seed. In Algorithm 1, we restrict recipes to contain at most 3 domains ($l_{max} = 3$) and set the number of iterations for each phase to be 5 and 10 ($i_{dom} = 5$ and $i_{set} = 10$).

The results are presented in Fig. 4, which shows the number of assertions that are verified with the best recipe found by each algorithm as well as by the default recipe. All algorithms outperform the default recipe for both usage scenarios, verifying almost twice as many assertions on average. The random-

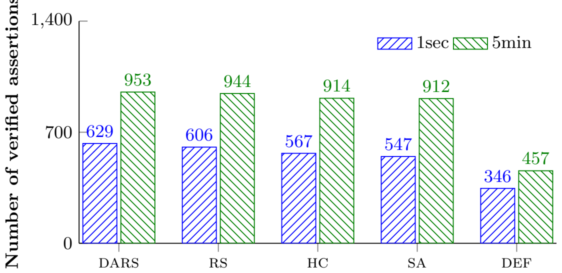


**Fig. 4.** Comparison of the number of assertions verified with the best recipe generated by each optimization algorithm and with the default recipe, for varying timeouts.

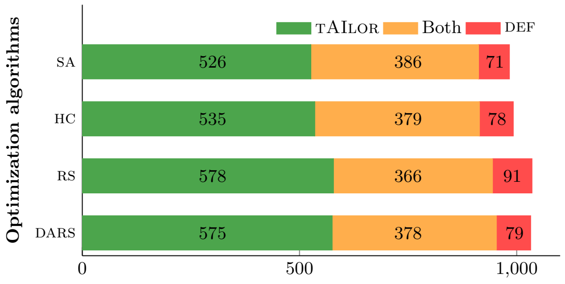


**Fig. 5.** Comparison of the number of assertions verified by a tailored vs. the default recipe.

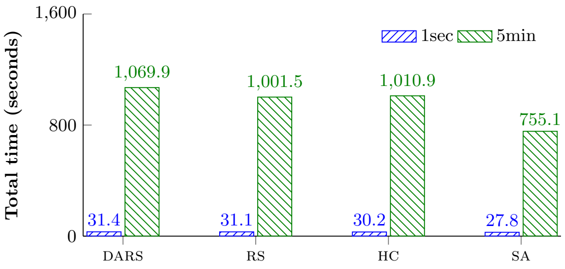


**Fig. 6.** Comparison of the total time (in sec) that each algorithm requires for all iterations, for varying timeouts.

sampling algorithms are shown to find better recipes than the others, with DARS being the most effective. Hill climbing is less effective since it gets stuck in local cost minima despite restarts. Simulated annealing is the least effective because it slowly climbs up the poset toward more precise domains (see Algorithm 2). However, as we explain later, we expect the algorithms to converge on the number of verified assertions for more iterations.

Figure 5 gives a more detailed comparison with the default recipe for the time limit of 5 min. In particular, each horizontal bar shows the total number of assertions verified by each algorithm. The orange portion represents the assertions verified by both the default recipe and the optimization algorithm, while the green and red portions represent the assertions only verified by the algorithm and default recipe, respectively. These results show that, in addition to verifying hundreds of new assertions, TAILOR is able to verify the vast majority of assertions proved by the default recipe, regardless of optimization algorithm.

In Fig. 6, we show the total time each algorithm takes for all iterations. DARS takes the longest. This is due to generating more precise recipes thanks to its domain knowledge. Such recipes typically take longer to run but verify more assertions (as in Fig. 4). On average, for all algorithms, TAILOR requires only 30 s to complete all iterations for the 1-s timeout and 16 min for the 5-min timeout. As discussed in Sect. 2, this tuning time can be spent offline.

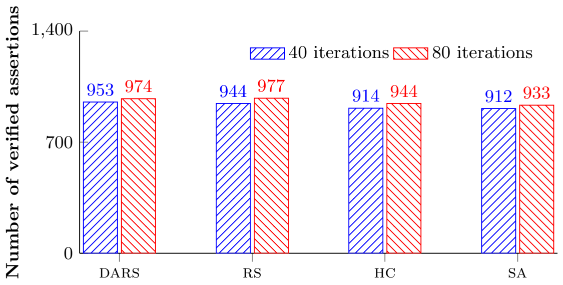


**Fig. 7.** Comparison of the number of assertions verified with the best recipe generated by the different optimization algorithms, for different numbers of iterations.

Figure 7 compares the total number of assertions verified by each algorithm when TAILOR runs for 40 ($i_{dom} = 5$ and $i_{set} = 10$) and 80 ($i_{dom} = 10$ and $i_{set} = 20$) iterations. The results show that only a relatively small number of additional assertions are verified with 80 iterations. In fact, we expect the algorithms to eventually converge on the number of verified assertions, given the time limit and precision of the available domains.

As DARS performs best in this comparison, we only evaluate DARS in the remaining research questions. We use a 5-min timeout.

**RQ1 takeaway:** TAILOR verifies between 1.6–2.1× the assertions of the default recipe, regardless of optimization algorithm, timeout, or number of iterations. In fact, even very simple algorithms (such as RS) significantly outperform the default recipe.

**RQ2: Are the Tailored Recipes Optimal?** To check the optimality of the tailored recipes, we compared them with the most precise (and least efficient) CRAB configuration. It uses the most precise domains from Fig. 3 (i.e., `bool`, `polyhedra`, `term(int)`, `ric`, `boxes`, and `term(disInt)`) in a recipe of 6 ingredients and assigns the most precise values to all other settings from Table 1. We generously gave a 30-min timeout to this recipe.

For 21 out of 120 files, the most precise recipe ran out of memory (264 GB). For 86 files, it terminated within 5 min, and for 13, it took longer (within 30 min)—in many cases, this was even longer than TAILOR's tuning time in Fig. 6. We compared the number of assertions verified by our tailored recipes (which do not exceed 5 min) and by the most precise recipe. For the 86 files that terminated within 5 min, our recipes prove 618 assertions, whereas the most precise recipe proves 534. For the other 13 files, our recipes prove 119 assertions, whereas the most precise recipe proves 98.

Consequently, our (in theory) less precise and more efficient recipes prove more assertions in files where the most precise recipe terminates. Possible explanations for this non-intuitive result are: (1) Polyhedra coefficients may overflow, in which case the constraints are typically ignored by abstract interpreters, and

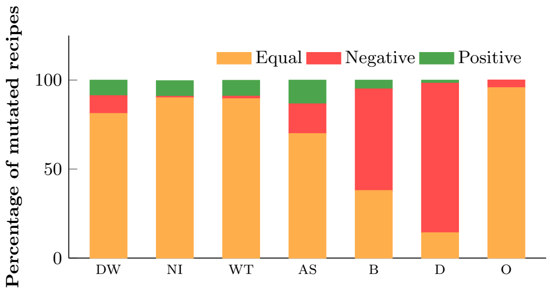


**Fig. 8.** Effect of different settings on the precision and performance of the abstract interpreter. (DW: `NUM_DELAY_WIDEN`, NI: `NUM_NARROW_ITERATIONS`, WT: `NUM_WIDEN_-THRESHOLDS`, AS: array smashing, B: backward analysis, D: abstract domain, O: ingredient ordering).

(2) more precise domains with different widening operations may result in less precise results [2,45].

We also evaluated the optimality of tailored recipes by mutating individual parts of the recipe and comparing to the original. In particular, for each setting in Table 1, we tried all possible values and replaced each domain with all other comparable domains in the poset of Fig. 3. For example, for a recipe including `zones`, we tried `octagons`, `polyhedra`, and `intervals`. In addition, we tried all possible orderings of the recipe ingredients, which in theory could produce different results. We observed whether these changes resulted in a difference in the precision and performance of the analyzer.

Figure 8 shows the results of this experiment, broken down by setting. Equal (in orange) indicates that the mutated recipe proves the same number of assertions within ±5 s of the original. Positive (in green) indicates that it either proves more assertions or the same number of assertions at least 5 s faster. Negative (in red) indicates that the mutated recipe either proves fewer assertions or the same number of assertions at least 5 seconds slower.

The results show that, for our benchmarks, mutating the recipe found by TAILOR rarely led to an improvement. In particular, at least 93% of all mutated recipes were either equal to or worse than the original recipe. In the majority of these cases, mutated recipes are equally good. This indicates that there are many optimal or close-to-optimal solutions and that TAILOR is able to find one.

**RQ2 takeaway:** As compared to the most precise recipe, TAILOR verified more assertions across benchmarks where the most precise recipe terminated. Furthermore, mutating recipes found by TAILOR resulted in improvement only for less than 7% of recipes.

**RQ3: How Diverse are the Tailored Recipes?** To motivate the need for optimization, we must show that tailored recipes are sufficiently diverse such that they could not be replaced by a well-crafted default recipe. To better understand the characteristics of tailored recipes, we manually inspected all of them.

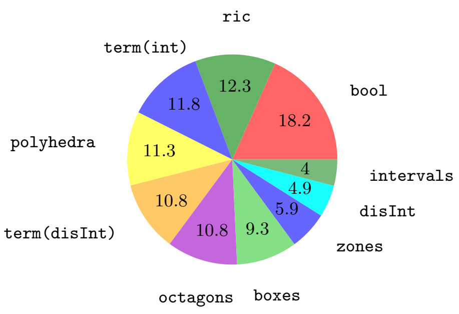


**Fig. 9.** Occurrence of domains (in %) in the best recipes for all assertion types.

tAIlor generated recipes of length greater than 1 for 61 files. Out of these, 37 are of length 2 and 24 of length 3. For 77% of generated recipes, `NUM_DELAY_-WIDEN` is not set to the default value of 1. Additionally, 55% of the ingredients enable array smashing, and 32% enable backward analysis.

Figure 9 shows how often (in percentage) each abstract domain occurs in a best recipe found by tAIlor. We observe that all domains occur almost equally often, with 6 of the 10 domains occurring in between 9% and 13% of recipes. The most common domain was `bool` at 18%, and the least common was `intervals` at 4%. We observed a similar distribution of domains even when instrumenting the benchmarks with only one assertion type, e.g., checking for integer overflow.

We also inspected which domain combinations are frequently used in the tailored recipes. One common pattern is combinations between `bool` and numerical domains (18 occurrences). Similarly, we observed 2 occurrences of `term(disInt)` together with `zones`. Interestingly, the less powerful variants of combining `disInt` with `zones` (3 occurrences) and `term(int)` with `zones` (6 occurrences) seem to be sufficient in many cases. Finally, we observed 8 occurrences of `polyhedra` or `octagons` with `boxes`, which are the most precise convex and non-convex domains. Our approach is, thus, not only useful for users, but also for designers of abstract interpreters by potentially inspiring new domain combinations.

**RQ3 takeaway:** The diversity of tailored recipes prevents replacing them with a single default recipe. Over half of the tailored recipes contain more than one ingredient, and ingredients use a variety of domains and their settings.

**RQ4: How Resilient are the Tailored Recipes to Code Changes?** We expect tailored recipes to be resilient to code changes, i.e., to retain their optimality across several changes without requiring re-tuning. We now evaluate if a recipe tailored for one code version is also tailored for another, even when the two versions are 50 commits apart.

For this experiment, we took a random sample of 60 files from our benchmarks and retrieved the 50 most recent commits per file. We only sampled 60 out of

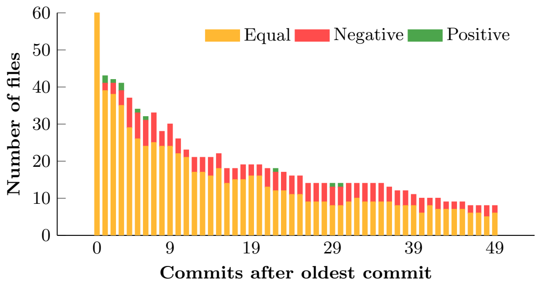


**Fig. 10.** Difference in the safe assertions across commits.

120 files as building these files for each commit is quite time consuming—it can take up to a couple of days. We instrumented each file version with the four assertion types described in Sect. 5.2. It should be noted that, for some files, we retrieved fewer than 50 versions either because there were fewer than 50 total commits or our build procedure for the project failed on older commits. This is also why we did not run this experiment for over 50 commits.

We analyzed each file version with the best recipe, $R_o$, found by tAIlor for the oldest file version. We compared this recipe with new best recipes, $R_n$, that were generated by tAIlor when run on each subsequent file version. For this experiment, we used a 5-min timeout and 40 iterations.

Note that, when running tAIlor with the same optimization algorithm and random seed, it explores the same recipes. It is, therefore, very likely that recipe $R_o$ for the oldest commit is also the best for other file versions since we only explore 40 different recipes. To avoid any such bias, we performed this experiment by seeding tAIlor with a different random seed for each commit. The results are shown in Fig. 10.

In Fig. 10, we give a bar chart comparing the number of files per commit that have a positive, equal, and negative difference in the number of verified assertions, where commit 0 is the oldest commit and 49 the newest. An equal difference (in orange) means that recipe $R_o$ for the oldest commit proves the same number of assertions in the current file version, $f_n$, as recipe $R_n$ found by running tAIlor on $f_n$. To be more precise, we consider the two recipes to be equal if they differ by at most 1 verified assertion or 1% of verified assertions since such a small change in the number of safe assertions seems acceptable in practice (especially given that the total number of assertions may change across commits). A positive difference (in green) means that $R_o$ achieves better verification results than $R_n$, that is, $R_o$ proves more assertions safe (over 1 assertion or 1% of the assertions that $R_n$ proves). Analogously, a negative difference (in red) means that $R_o$ proves fewer assertions. We do not consider time here because none of the recipes timed out when applied on any file version.

Note that the number of files decreases for newer commits. This is because not all files go forward by 50 commits, and even if they do, not all file versions build. However, in a few instances, the number of files increases going forward in time. This happens for files that change names, and later, change back, which we do not account for.

For the vast majority of files, using recipe $R_o$ (found for the oldest commit) is as effective as using $R_n$ (found for the current commit). The difference in safe assertions is negative for less than a quarter of the files tested, with the average negative difference among these files being around 22% (i.e., $R_o$ proved 22% fewer assertions than $R_n$ in these files). On the remaining three quarters of the files tested however, $R_o$ proves at least as many assertions as $R_n$, and thus, $R_o$ tends to be tailored across code versions.

Commits can result in both small and large changes to the code. We therefore also measured the average difference in the number of verified assertions per changed line of code with respect to the oldest commit. For most files, regardless of the number of changed lines, we found that $R_o$ and $R_n$ are equally effective, with changes to 1000 LOC or more resulting in little to no loss in precision. In particular, the median difference in safe assertions across all changes between $R_o$ and $R_n$ was 0 (i.e., $R_o$ proved the same number of assertions safe as $R_n$), with a standard deviation of 15 assertions. We manually inspected a handful of outliers where $R_o$ proved significantly fewer assertions than $R_n$ (difference of over 50 assertions). These were due to one file from GIT where $R_o$ is not as effective because the widening and narrowing settings have very low values.

**RQ4 takeaway:** For over 75% of files, TAILOR's recipe for a previous commit (from up to 50 commits previous) remains tailored for future versions of the file, indicating the resilience of tailored recipes across code changes.

## 5.4 Threats to Validity

We have identified the following threats to the validity of our experiments.

**Benchmark Selection.** Our results may not generalize to other benchmarks. However, we selected popular GitHub projects from different application domains (see Table 2). Hence, we believe that our benchmark selection mitigates this threat and increases generalizability of our findings.

**Abstract Interpreter and Recipe Settings.** For our experiments, we only used a single abstract interpreter, CRAB, which however is a mature and actively supported tool. The selection of recipe settings was, of course, influenced by the available settings in CRAB. Nevertheless, CRAB implements the generic architecture of Fig. 2, used by most abstract interpreters, such as those mentioned at the beginning of Sect. 3. We, therefore, expect our approach to generalize to such analyzers.

**Optimization Algorithms.** We considered four optimization algorithms, but in Sect. 4.3, we explain why these are suitable for our application domain. Moreover, TAILOR is configurable with respect to the optimization algorithm.

**Assertion Types.** Our results are based on four types of assertions. However, these cover a wide range of runtime errors that are commonly checked by static analyzers.

## 6 Related Work

The impact of different abstract-interpretation configurations has been previously evaluated [54] for Java programs and partially inspired this work. To the best of our knowledge, we are the first to propose tailoring abstract interpreters to custom usage scenarios using optimization.

However, optimization is a widely used technique in many engineering disciplines. In fact, it is also used to solve the general problem of algorithm configuration [31], of which there exist numerous instantiations, for instance, to tune hyper-parameters of learning algorithms [3,18,52] and options of constraint solvers [32,33]. Existing frameworks for algorithm configuration differ from ours in that they are not geared toward problems that are solved by sequences of algorithms, such as analyses with different abstract domains. Even if they were, our experience with tAIlor shows that there seem to be many optimal or close-to-optimal configurations, and even very simple optimization algorithms such as rs are surprisingly effective (see RQ2); similar observations were made about the effectiveness of random search in hyper-parameter tuning [4].

In the rest of this section, we focus on the use of optimization in program analysis. It has been successfully applied to a number of program-analysis problems, such as automated testing [19,20], invariant inference [50], and compiler optimizations [49].

Recently, researchers have started to explore the direction of enriching program analyses with machine-learning techniques, for example, to automatically learn analysis heuristics [27,34,47,51]. A particularly relevant body of work is on adaptive program analysis [28–30], where existing code is analyzed to learn heuristics that trade soundness for precision or that coarsen the analysis abstractions to improve memory consumption. More specifically, adaptive program analysis poses different static-analysis problems as machine-learning problems and relies on Bayesian optimization to solve them, e.g., the problem of selectively applying unsoundness to different program components (e.g., different loops in the program) [30]. The main insight is that program components (e.g., loops) that produce false positives are alike, predictable, and share common properties. After learning to identify such components for existing code, this technique suggests components in unseen code that should be analyzed unsoundly.

In contrast, tAIlor currently does not adjust soundness of the analysis. However, this would also be possible if the analyzer provided the corresponding configurations. More importantly, adaptive analysis focuses on learning analysis heuristics based on existing code in order to generalize to arbitrary, unseen code. tAIlor, on the other hand, aims to tune the analyzer configuration to a custom usage scenario, including a particular program under analysis. In addition, the custom usage scenario imposes user-specific resource constraints, for instance by limiting the time according to a phase of the software-engineering life cycle. As we show in our experiments, the tuned configuration remains tailored to several versions of the analyzed program. In fact, it outperforms configurations that are meant to generalize to arbitrary programs, such as the default recipe.

## 7 Conclusion

In this paper, we have proposed a technique and framework that tailors a generic abstract interpreter to custom usage scenarios. We instantiated our framework with a mature abstract interpreter to perform an extensive evaluation on real-world benchmarks. Our experiments show that the configurations generated by tAIlor are vastly better than the default options, vary significantly depending on the code under analysis, and typically remain tailored to several subsequent code versions. In the future, we plan to explore the challenges that an inter-procedural analysis would pose, for instance, by using a different recipe for computing a summary of each function or each calling context.

**Acknowledgements.** We are grateful to the reviewers for their constructive feedback. This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science). Jorge Navas was supported by NSF grant 1816936.

## References

1. The BDDAPRON logico-numerical abstract domains library. http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron
2. Amato, G., Rubino, M.: Experimental evaluation of numerical domains for inferring ranges. ENTCS **334**, 3–16 (2018)
3. Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for hyper-parameter optimization. In: NIPS, pp. 2546–2554 (2011)
4. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. JMLR **13**, 281–305 (2012)
5. Blanchet, B., et al.: A static analyzer for large safety-critical software. In: PLDI, pp. 196–207. ACM (2003)
6. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: a framework for static analysis based on abstract interpretation. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 271–277. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10431-7_20
7. Calcagno, C., Distefano, D.: Infer: an automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33
8. Calcagno, C., et al.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 3–11. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_1
9. Chang, B.-Y.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 147–163. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_11

10. Christakis, M., Bird, C.: What developers want and need from program analysis: an empirical study. In: ASE, pp. 332–343. ACM (2016)
11. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: ISOP, pp. 106–130. Dunod (1976)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
13. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. JLP **13**, 103–179 (1992)
14. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55844-6_142
15. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Autom. Softw. Eng. **6**, 69–95 (1999)
16. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, pp. 84–96. ACM (1978)
17. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_2
18. Falkner, S., Klein, A., Hutter, F.: BOHB: robust and efficient hyperparameter optimization at scale. In: ICML. PMLR, vol. 80, pp. 1436–1445. PMLR (2018)
19. Fu, Z., Su, Z.: Mathematical execution: a unified approach for testing numerical code. CoRR abs/1610.01133 (2016)
20. Fu, Z., Su, Z.: Achieving high coverage for floating-point code via unconstrained programming. In: PLDI, pp. 306–319. ACM (2017)
21. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: An abstract domain of uninterpreted functions. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 85–103. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_4
22. Gershuni, E., et al.: Simple and precise static analysis of untrusted Linux kernel extensions. In: PLDI, pp. 1069–1084. ACM (2019)
23. Granger, P.: Static analysis of arithmetical congruences. Int. J. Comput. Math. **30**, 165–190 (1989)
24. Gurfinkel, A., Chaki, S.: Boxes: a symbolic abstract domain of boxes. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 287–303. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_18
25. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
26. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of C/C++ programs. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 148–168. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_8
27. Heo, K., Oh, H., Yang, H.: Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 237–256. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_12
28. Heo, K., Oh, H., Yang, H.: Resource-aware program analysis via online abstraction coarsening. In: ICSE, pp. 94–104. IEEE Computer Society/ACM (2019)

29. Heo, K., Oh, H., Yang, H., Yi, K.: Adaptive static analysis via learning with Bayesian optimization. TOPLAS **40**, 14:1–14:37 (2018)
30. Heo, K., Oh, H., Yi, K.: Machine-learning-guided selectively unsound static analysis. In: ICSE, pp. 519–529. IEEE Computer Society/ACM (2017)
31. Hutter, F.: Automated Configuration of Algorithms for Solving Hard Computational Problems. Ph.D. thesis, The University of British Columbia, Canada (2009)
32. Hutter, F., Babic, D., Hoos, H.H., Hu, A.J.: Boosting verification by automatic tuning of decision procedures. In: FMCAD, pp. 27–34. IEEE Computer Society (2007)
33. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: AAAI, pp. 1152–1157. AAAI (2007)
34. Jeong, S., Jeon, M., Cha, S.D., Oh, H.: Data-driven context-sensitivity for points-to analysis. PACMPL **1**, 100:1–100:28 (2017)
35. Karr, M.: Affine relationships among variables of a program. Acta Inf. **6**, 133–151 (1976)
36. Kirkpatrick, S., Gelatt, C.D., Jr., Vecchi, M.P.: Optimization by simulated annealing. Science **220**, 671–680 (1983)
37. Lakhdar-Chaouch, L., Jeannet, B., Girault, A.: Widening with thresholds for programs with complex control graphs. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 492–502. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_38
38. Mátyáš, I.: Random optimization. Avtomat. i Telemekh. **26**, 246–253 (1965)
39. Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., Teller, E.: Equation of state calculations by fast computing machines. J. Chem. Phys. **21**, 1087–1092 (1953)
40. Mihaila, B., Sepp, A., Simon, A.: Widening as abstract domain. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 170–184. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38088-4_12
41. Miné, A.: A few graph-based relational numerical abstract domains. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 117–132. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45789-5_11
42. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: LCTES, pp. 54–63. ACM (2006)
43. Miné, A.: The Octagon abstract domain. HOSC **19**, 31–100 (2006)
44. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 348–363. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_23
45. Monniaux, D., Le Guen, J.: Stratified static analysis based on variable dependencies. ENTCS **288**, 61–74 (2012)
46. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for C-like languages. In: PLDI, pp. 229–238. ACM (2012)
47. Raychev, V., Vechev, M.T., Krause, A.: Predicting program properties from 'big code'. CACM **62**, 99–107 (2019)
48. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Pearson Education (2010)
49. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: ASPLOS, pp. 305–316. ACM (2013)
50. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: CAV. LNCS, vol. 8559, pp. 88–105. Springer (2014)

51. Singh, G., Püschel, M., Vechev, M.: Fast numerical program analysis with reinforcement learning. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 211–229. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_12
52. Thornton, C., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In: KDD, pp. 847–855. ACM (2013)
53. Venet, A., Brat, G.P.: Precise and efficient static array bound checking for large embedded C programs. In: PLDI, pp. 231–242. ACM (2004)
54. Wei, S., Mardziel, P., Ruef, A., Foster, J.S., Hicks, M.: Evaluating design tradeoffs in numeric static analysis for Java. In: ESOP. LNCS, vol. 10801, pp. 653–682. Springer (2018)

# Functional Correctness of C Implementations of Dijkstra's, Kruskal's, and Prim's Algorithms

Anshuman Mohan(✉), Wei Xiang Leow, and Aquinas Hobor

School of Computing, National University of Singapore, Singapore, Republic of Singapore
amohan@cs.cornell.edu

**Abstract.** We develop machine-checked verifications of the full functional correctness of C implementations of the eponymous graph algorithms of Dijkstra, Kruskal, and Prim. We extend Wang *et al.*'s CertiGraph platform to reason about labels on edges, undirected graphs, and common spatial representations of edge-labeled graphs such as adjacency matrices and edge lists. We certify binary heaps, including Floyd's bottom-up heap construction, heapsort, and increase/decrease priority.

Our verifications uncover subtle overflows implicit in standard textbook code, including a nontrivial bound on edge weights necessary to execute Dijkstra's algorithm; we show that the intuitive guess fails and provide a workable refinement. We observe that the common notion that Prim's algorithm requires a connected graph is wrong: we verify that a standard textbook implementation of Prim's algorithm can compute minimum spanning forests without finding components first. Our verification of Kruskal's algorithm reasons about two graphs simultaneously: the undirected graph undergoing MSF construction, and the directed graph representing the forest inside union-find. Our binary heap verification exposes precise bounds for the heap to operate correctly, avoids a subtle overflow error, and shows how to recycle keys to avoid overflow.

**Keywords:** Separation logic · Graph algorithms · Coq · VST

## 1 Introduction

Dijkstra's eponymous shortest-path algorithm [22] finds the cost-minimal paths from a distinguished *source* vertex to all reachable vertices in a directed graph. Prim's [61] and Kruskal's [42] algorithms return minimal spanning trees for undirected graphs. Binary heaps are the first priority queue one typically encounters. These algorithms/structures are classic and ubiquitous, appearing widely in textbooks [20,33,36,65,66,68] and in real routing protocol libraries.

In addition to decades of use and textbook analysis, recent efforts have verified one or more of these algorithms in proof assistants and formally proved

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 801–826, 2021.
https://doi.org/10.1007/978-3-030-81688-9_37

claims about their behavior [12,15,30,45,53]. A reasonable person might think that all that can be said, has been. However, we have found that textbook code glosses over a cornucopia of issues that routinely crop up in real-world settings: under/overflows, integration with performant data structures, manual memory (de-)allocation, error handling, casts, memory alignment, *etc.* Further, previous verification efforts with formal checkers often operate within idealized formal environments, which likewise leads them to ignore the same kinds of issues.

In our work, we provide C implementations of each of these algorithms/data structures, and prove in Coq [71] the functional correctness of the same with respect to the formal semantics of CompCert C [50]. By "functional correctness" we mean natural algorithmic specifications; we do not prove resource bounds. Although our C code is developed from standard textbooks, we uncover several subtleties that are absent from the algorithmic and formal methods literature:

§3.2 an overflow in Dijkstra's algorithm, avoiding which requires a nontrivial refinement to the algorithm's precondition to bound edge weights;
§4.2 that the specification of Prim's algorithm can be improved to apply to disconnected graphs without any change to textbook (pseudo-)code;
§4.2 the presence of a wholly unneeded line of (pseduo-)code in Prim's algorithm, and an associated unneeded function argument;
§5 several potential overflows in binary heaps equipped with Floyd's linear-time build-heap function and an edit-priority operation.

We wish to develop general and reusable techniques for verifying graph-manipulating programs written in real programming languages. This is a significant challenge, and so we choose to leverage and/or extend three large existing proof developments to state and prove the full functional correctness of our code in Coq: CompCert; the Verified Software Toolchain [4] (VST) separation logic [59] deductive verifier; and our own previous efforts [73], hereafter dubbed the CertiGraph project. Our primary extensions are to the third, and include:

§2.1 pure/abstract reasoning for graphs with edge labels, (*e.g.*, a distinguished edge-label value for "infinity" that indicates invalid/absent edges);
§2.2 spatial representations and associated reasoning for edge-labeled graphs (several flavors of adjacency matrices as well as edge lists);
§2.3 pure reasoning for undirected graphs (*e.g.*, notions of connectedness).

We prove that our pure machinery and our spatial machinery are well-isolated from each other by verifying several implementations (of each of Dijkstra and Prim) that represent graphs differently in memory but reuse the entire pure portion of the proof. Likewise, we show that our spatial reasoning is generic by reusing graph representations across Dijkstra and Prim. Our verification of Kruskal proves that we can reason about two graphs simultaneously: a directed graph with vertex labels for union-find and an undirected graph with edge labels for which we are building a spanning forest. In addition to our verification of

Dijkstra, Prim, and Kruskal, we develop increased lemma support for the preexisting CertiGraph union-find example [73]. Our extension to "base VST" (*e.g.*, verifications without graphs) primarily consists of our verified binary heap.

The remainder of this paper is organized as follows:

- §2 We explain our extensions to CertiGraph: edge-labeled graphs, spatial representations of such graphs, and undirected graphs.
- §3 We explain our verification of Dijkstra's algorithm in some detail, discuss a potential overflow, and refine the precondition to avoid it.
- §4 We overview our verifications of the Minimum Spanning Tree/Forest algorithms of Prim and Kruskal, focusing on high-level points such as our improved novel specification of Prim's.
- §5 We overview our verification of binary heaps, including a discussion of Floyd's bottom-up heap construction and the `edit_priority` operation.
- §6 We briefly discuss engineering, *e.g.* statistics for our formal development.
- §7 We discuss related work, outline future research directions, and conclude.

Our results are completely machine-checked in Coq and publicly available [1].

## 2 Extensions to CertiGraph

We begin with the briefest of introductions to CertiGraph's core structure and then detail the extensions we make to various levels of CertiGraph in service of our Dijkstra, Prim, and Kruskal verifications. Ignoring modularity and eliding elements not used in this work, a mathematical graph in CertiGraph is a tuple: ($\mathcal{V}$, $\mathcal{E}$, `vvalid`, `evalid`, `src`, `dst`, `vlabel`, `elabel`, `sound`). Here $\mathcal{V}/\mathcal{E}$ are the carrier types of vertices/edges, `vvalid`/`evalid` place restrictions specifying whether a vertex/edge is valid[1], and `src`/`dst` : $\mathcal{E} \to \mathcal{V}$ map edges to their source/destination. Labels are allowed on vertices and edges, and a `sound`ness condition allows custom application-specific restrictions [73]. Mathematical graphs connect to graphs in computer memory via spatial predicates in separation logic.

### 2.1 Pure Reasoning for Adjacency Matrix-Represented Graphs

Two of our algorithms operate over graphs represented as adjacency matrices. Not every legal graph can be represented as an adjacency matrix, so we develop a unified, reusable, and extendable `sound`ness condition `SoundAdjMat` that a graph must satisfy in order for it to be represented as an adjacency matrix.

`SoundAdjMat` is parameterized by the graph's `size` and a distinguished number `inf`. We restrict most fields in the tuple: ($\mathcal{V} = \mathbb{Z}$, $\mathcal{E} = \mathbb{Z} \times \mathbb{Z}$, `vvalid` $= \lambda v.\ 0 \le v <$ `size`, `evalid` $= \ldots$, `src` $= \mathit{fst}$, `dst` $= \mathit{snd}$, `vlabel`, `elabel`, `sound` $= \ldots$). We also restrict the carrier type of vertex labels to `unit`

[1] Validity denotes presence in the graph: *e.g.*, if we are using $\mathbb{Z}$ as the carrier type $\mathcal{V}$, and have only 7 vertices, then `vvalid`$(x)$ is probably the proposition $0 \le x < 7$).

and edge labels to $\mathbb{Z}$. We require the parameters `size` and `inf` be strictly positive and representable on the machine. Most critical, however, is the semantics of `evalid`: a valid edge must have a machine-representable label and that label cannot have value `inf`; an invalid edge *must* have label `inf`. Last, the graph must be finite.

The restriction on edge labels is necessary because we are working with labeled adjacency matrices on a real system: we need to set aside a distinguished number `inf` such that edgeweight `inf` indicates the *absence* of an edge. We cannot prescribe some `inf` because client needs can vary widely. For instance, our verifications of Dijkstra's and Prim's algorithms require subtly different `inf`s.

`SoundAdjMat` guarantees spatial representability as an adjacency matrix, but it can be extended with further algorithm-specific restrictions before being plugged in for `sound`. Dijkstra's algorithm requires nonnegative edge weights, and—as we will discuss in §3.2—nontrivial restrictions on `size` and `inf`.

## 2.2 New Spatial Representations for Edge-Labeled Graphs

We give predicates for adjacency matrices and edge lists for edge-labeled graphs.

**Adjacency Matrices.** Adjacency matrices enable efficient label access for edge-labeled graphs. We support three common adjacency matrix representations: a stack-allocated 2D array `int graph[size][size]`, a stack-allocated 1D array `int graph[size×size]`, and a heap-allocated 2D array `int **graph`. To the casual observer, these are essentially interchangeable, but that is a mistake when thinking spatially. Apart from the arithmetic that the second flavor uses to access cells, there is a more subtle point: the first and second enjoy a contiguous block of memory, but the third does not: it is an allocated "spine" with pointers to separately-allocated rows. For a taste, the spatial representation of the first is:

$$\begin{aligned}
\mathit{arr\_addr}(\mathit{ptr}, i, \texttt{size}) &\triangleq \mathit{ptr} + (i \times \texttt{size})\\
\mathsf{array}(\mathit{ptr}, \mathit{list}) &\triangleq \mathop{\huge\ast}_{i \in [0, |\mathit{list}|)} (\mathit{ptr} + i) \mapsto \mathit{list}[i])\\
\mathsf{arr\_rep}(\gamma, i, \mathit{ptr}) &\triangleq \texttt{let}\ \mathit{row}\ \texttt{:=}\ \mathsf{graph2mat}(\gamma)[i]\ \texttt{in}\\
&\qquad \mathsf{array}(\mathit{arr\_addr}(\mathit{ptr}, i, |\mathit{row}|), \mathit{row})\\
\mathsf{graph\_rep}(\gamma, \mathit{g\_addr}, \_) &\triangleq \mathop{\huge\ast}_{v \in \gamma} \mathsf{arr\_rep}(\gamma, v, \mathit{g\_addr})
\end{aligned}$$

We use the separation logic $*$ in its iterated form to say that the arrays are separate in memory. We elide details relating to object sizes, pointer alignment, and so forth, although our formal proofs handle such matters. Of particular note are graph2mat, which performs two projections to drag out the graph's nested edge labels into a 2D matrix, and *arr_addr*, which in this instance simply computes the address of any legal row $i$ from the base address of the graph. Notice that this graph_rep predicate ignores its third argument. To represent a heap-allocated 2D array we can still use graph2mat but can no longer use address arithmetic; the third parameter is then a list of pointers to the row sub-arrays.

While ironing out these spatial wrinkles, we develop utilities that easily unfold and refold our adjacency matrices, thus smoothing user experience when reading and writing arrays and cells. Of course these utilities themselves vary by flavor of representation, but the net effect is that users of our adjacency matrices really can be agnostic to the style of representation they are using (see §3.1).

**Edge Lists.** Edge lists are the representation of choice for sparse graphs. Our C implementation defines an `edge` as a `struct` containing `src`, `dst`, and `weight`, and defines a `graph` as a `struct` containing the graph's size, edge count, and an array of `edges`. Our spatial representation follows this pattern:

$$\mathsf{graph\_rep}(\gamma, g\_addr, e\_addr) \triangleq$$
$$\big(g\_addr \mapsto (|\gamma.V|, |\gamma.E|, e\_addr)\big) * \mathsf{array}(e\_addr, \gamma.E)$$

## 2.3 Undirectedness in a Directed World

The CertiGraph library presented in [73] supports only directed graphs, and, as we have seen, bakes direction-reliant idioms such as `src` and `dst` deep into its development. Our challenge is to add support for undirected graphs atop of this.

Our approach is to observe that every directed graph can be treated as an undirected graph by ignoring edge direction. We develop a lightweight layer of "undirected flavored" definitions atop of the existing "directed flavored" definitions, state and prove connections between these, and then build the undirected infrastructure we need. The result is that we retain full access to CertiGraph's graph theory formalizations modulo some mathematical bridging.

Our basic "undirected flavored" definitions are standard [20]. Vertices $u$ and $v$ are `adjacent` if there is an edge between them in either direction; vertices are self-adjacent. A valid `upath` (undirected path) is list of valid vertices that form a pairwise-adjacent chain. Two vertices are `connected` when a valid `upath` features them as head and foot (essentially the transitive closure of `adjacenct`).

The definitions above sync up with preexisting "directed flavored" definitions. Intuitively, undirectedness is more lax than directedness, and so it is unsurprising that these connections are straightforward weakenings of directed properties. We next give standard definitions [20] that culminate in `minimum_spanning_forest`, which is exactly our postcondition of both Prim's and Kruskal's algorithms.[2]

An undirected cycle (`ucycle`) is a valid non-empty `upath` whose first and last vertices are equal. A `connected_graph` means that any two valid vertices are `connected`. `is_partial_graph f g` means everything in `f` is in `g`. We proceed:

```
Definition uforest g :=
 (∀ e, evalid g e → strong_evalid g e) ∧
 (∀ p l, ¬ucycle g p l).
Definition spanning g g' :=
 ∀ u v, connected g u v ↔ connected g' u v.
```

[2] That Prim's postcondition has a *forest* may raise an eyebrow. See §4.2.

```
Definition spanning_uforest f g :=
  is_partial_graph f g ∧ uforest f ∧ spanning f g.
```

The `strong_evalid` predicate means that the `src` and `dst` of the edge are also valid, so *e.g.*, a valid edge cannot point to a deleted/absent vertex. The second conjunct of `uforest` is critical: a forest has no undirected cycles. The other definitions are straightforward from there, and `minimum_spanning_forest f g` means that no other spanning forest has lower total edge cost than `f`.

Our undirected work is also compatible with our new developments in §2.1 and §2.2. An adjacency matrix-representable undirected graph has all the pure properties discussed in `SoundAdjMat`, and also has symmetry across the left diagonal. We extend `SoundAdjMat` into `SoundUAdjMat` by requiring that all valid edges have `src` ≤ `dst`. This effectively "turns off" the matrix on one half of the diagonal and avoids double-counting. Prim's algorithm uses `SoundUAdjMat` and places no further restrictions. Further, spatial representations and fold/unfold utilities are shared across directed and undirected adjacency matrices.

## 3 Shortest Path

We verify a standard C implementation of Dijkstra's algorithm. We first sketch our proof in some detail with an emphasis on our loop invariants, then uncover and remedy a subtle overflow bug, and finish with a discussion of related work.

### 3.1 Verified Dijkstra's Algorithm in C

Figure 1 shows the code and proof sketch of Dijkstra's algorithm. Red text is used in the figure to highlight changes compared to the annotation immediately prior. Our code is implemented exactly as suggested by CLRS [20], so we refer readers there for a general discussion of the algorithm. The adjacency-matrix-represented graph $\gamma$ of `size` vertices is passed as the parameter `g` along with the source vertex `src` and two allocated arrays `dist` and `prev`. The spatial predicate $\mathsf{array}(\mathtt{x}, \boldsymbol{v})$, which connects an array pointer `x` with its contents $\boldsymbol{v}$, is standard and unexciting. $\mathsf{PQ}(\mathsf{pq}, \mathit{heap})$ is the spatial representation of our priority queue (PQ) and $\mathsf{Item}(\mathtt{i}, (\mathit{key}, \mathit{pri}, \mathit{data}))$ lays out a struct that we use to interact with the PQ; we leave the management of the PQ to the operations described in§ 5. Of greater interest is $\mathsf{AdjMat}(\mathtt{g}, \gamma)$, which as explained in §2.2, links the concrete memory values of `g` to an abstract mathematical graph $\gamma$, which in turn exposes an interface in the language of graph theory (*e.g.*, vertices, edges, labels). Graph $\gamma$ contains the general adjacency matrix restrictions given in §2.1 along with some further Dijkstra-specific restrictions to be explained in §3.2. We verify Dijkstra three times using different adjacency-matrix representations as explained in §2.2. Thanks to some careful engineering, the C code and the Coq verification are both almost completely agnostic to the form of representation. The only variation between implementations is when reading a cell (line 15), so we refactor this out into a straightforward helper method and verify it separately; accordingly, the proof bases for the three variants differ by less than 1%.

```
 1 void dijkstra (int **g, int src, int *dist,
 2               int *prev, int size, int inf {
 3 // $\{\mathsf{AdjMat}(g,\gamma) * \mathsf{array}(dist,\_) * \mathsf{array}(prev,\_) \land src \in \gamma \land connected(\gamma, src)\}$
 4  Item* temp = (Item*) mallocN(sizeof(Item));
 5  int* keys = mallocN (size * sizeof (int));
 6  PQ* pq = pq_make(size); int i, u, cost;
 7  for (i = 0; i < size; i++)
 8  { dist[i] = inf; prev[i] = inf; keys[i] = pq_push(pq,inf,i); }
 9  dist[src]= 0; prev[src]= src; pq_edit_priority(pq,keys[src],0);
10  while (pq_size(pq) > 0) {
11 // $\left\{\begin{array}{l} \exists dist, prev, popped, heap.\ \mathsf{AdjMat}(g,\gamma) * \mathsf{PQ}(pq, heap) * \mathsf{Item}(temp,\_) * \\ \mathsf{array}(dist, dist) * \mathsf{array}(prev, prev) * \mathsf{array}(keys, keys) \land \\ linked\_correctly(\gamma, heap, keys, dist, popped) \land \\ dijk\_correct(\gamma, src, popped, prev, dist) \end{array}\right\}$
12   pq_pop(pq, temp); u = temp->data;
13   for (i = 0; i < size; i++) {
14 // $\left\{\begin{array}{l} \exists dist', prev', heap'.\ \mathsf{AdjMat}(g,\gamma) * \mathsf{PQ}(pq, heap') * \\ \mathsf{array}(dist, dist') * \mathsf{array}(prev, prev') * \mathsf{array}(keys, keys) * \\ \mathsf{Item}(temp, (keys[u], dist[u], u)) \land min(dist[u], heap') \land \\ linked\_correctly(\gamma, heap', keys, dist', popped \uplus \{u\}) \land \\ dijk\_correct\_weak(\gamma, src, popped \uplus \{u\}, prev', dist', i, u) \end{array}\right\}$
15    cost = getCell(g, u, i);
16    if (cost < inf) {
17     if (dist[i] > dist[u] + cost) {
18      dist[i] = dist[u] + cost; prev[i] = u;
19      pq_edit_priority(pq, keys[i], dist[i]);
20   }}}} // $\left\{\begin{array}{l} \exists dist'', prev''.\ \mathsf{AdjMat}(g,\gamma) * \mathsf{PQ}(pq, \emptyset) * \mathsf{Item}(temp,\_) * \\ \mathsf{array}(dist, dist'') * \mathsf{array}(prev, prev'') * \mathsf{array}(keys, keys) \land \\ \forall dst.\ dst \in \gamma \to inv\_popped(\gamma, src, \gamma.V, prev'', dist'', dst) \end{array}\right\}$
21  freeN (temp); pq_free (pq); freeN (keys); return; }
```

**Fig. 1.** C code and proof sketch for Dijkstra's algorithm.

Dijkstra's algorithm uses a PQ to greedily choose the cheapest unoptimized vertex on line 12. The best-known distances to vertices are expected to improve as various edges are relaxed, and such improvements need to be logged in the PQ: Dijkstra's algorithm implicitly assumes that its PQ supports the additional operation `decrease_priority`. Our "advanced" PQ (§5.3) supports this operation in logarithmic time with the `pq_edit_priority` function[3].

The first nine lines are standard setup. The *keys* array, assigned on line 8, is thereafter a mathematical constant. The pure predicate *linked_correctly* contains the plumbing connecting the various mathematical arrays. The verification turns on the loop invariants on lines 11 and 14. The pure `while` invariant

[3] Because `decrease_priority` is relatively complex to implement, several popular workarounds (*e.g.* [12]) use simpler PQs at the cost of decreased performance.

$dijk\_correct(\gamma, src, popped, prev, dist)$ essentially unfolds into:

$$\begin{aligned}\forall dst.\ dst \in \gamma \rightarrow\ & inv\_popped(\gamma, src, popped, prev, dist, dst) \wedge \\ & inv\_unpopped(\gamma, src, popped, prev, dist, dst) \wedge \\ & inv\_unseen(\gamma, src, popped, prev, dist, dst)\end{aligned}$$

That is, a destination vertex *dst* falls into one of three categories:

1. *inv_popped*: if $dst \in popped$, then *dst* has been fully processed, *i.e.*, *dst* is reachable from *src* via a globally-optimal path *p* whose vertices are all in *popped*. Path *p* has been logged in *prev* and *p*'s cost is given in *dist*.
2. *inv_unpopped*: if $dst \notin popped$, but its known *dist*ance is less than `inf`, then *dst* is reachable in one step from a popped vertex *mom*. This route is locally optimal: we cannot improve the cost via an alternate popped vertex. Moreover, *prev* logs *mom* as the best-known way to reach *dst*, and *dist* logs the path cost via *mom* as the best-known cost.
3. *inv_unseen*: if $dst \notin popped$ and its known *dist*ance is `inf`, then there is no edge from any *popped* vertex to *dst*; in other words, *dst* is located deeper in the graph than has yet been explored.

After line 12, the above invariant is no longer true: a minimum-cost item *u* has been popped from the PQ, and so the *dist* and *prev* arrays need to be updated to account for this pop. The `for` loop does exactly this repair work. Its pure invariant $dijk\_correct\_weak(\gamma, src, popped, prev, dist, u, i)$ essentially unfolds into:

$$\begin{aligned}&\big(\forall dst.\ dst \in \gamma && \rightarrow inv\_popped(\gamma, src, popped, prev, dist, dst)\big) \wedge \\ &\big(\forall dst.\ 0 \le dst < i && \rightarrow inv\_unpopped(\gamma, src, popped, prev, dist, dst) \wedge \\ & && \quad\ inv\_unseen(\gamma, src, popped, prev, dist, dst)\big) \wedge \\ &\big(\forall dst.\ i \le dst < \texttt{size} && \rightarrow inv\_unpopped\_weak(\gamma, src, popped, prev, dist, dst, u) \wedge \\ & && \quad\ inv\_unseen\_weak(\gamma, src, popped, prev, dist, dst, u)\big)\end{aligned}$$

We now have five cases, many of which are familiar from *dijk_correct*:

1. *inv_popped*: as before; if $dst \in popped$, then it has been fully processed. For all "previously-popped vertices" (*i.e.,* except for *u*), this is trivial from *dijk_correct*. For *u* itself, we reach the heart of Dijkstra's correctness: the locally-optimal path to the cheapest unpopped vertex is *globally* optimal.
2. *inv_unpopped* (less than *i*): as before; if *dst* is reachable in one hop from a popped vertex *mom*, where now *mom* could be *u*. Initially this is trivial since $i = 0$, and we restore it as *i* increments by updating costs when they can be improved, as on lines 18 and 19.
3. *inv_unseen* (less than *i*): as before; some previously unseen neighbors of *u* may be transferred to unpopped status. This is also restored as *i* increments.
4. *inv_unpopped_weak* (between *i* and `size`): if *dst* is reachable in one hop from a previously-popped vertex *mom*, with potentially further improvements possible via *u*. As *i* increments, we strengthen it into *inv_unpopped* after considering whether routing via *u* improves the best-known cost to *dst*.

5. *inv_unseen_weak* (between $i$ and `size`): no edge exists from any previously-popped vertex to *dst*, but there may be one from $u$. As $i$ increments, we consider whether routing via $u$ reveals a path to *dst*. This is strengthened into *inv_unpopped* if so, and into *inv_unseen* if not.

At the end of the `for` loop the fourth and fifth cases fall away ($i$ = size), and the PQ and the *dist* and *prev* arrays finish "catching up" to the pop on line 12. This allows us to infer the `while` invariant *dijk_correct*, and thus continue the `while` loop. The `while` loop itself breaks when all vertices have been popped and processed. The second and third clauses of the `while` loop invariant *dijk_correct* then fall away, as seen on line 20: all vertices satisfy *inv_popped*, and are either optimally reachable or altogether unreachable. We are done.

## 3.2 Overflow in Dijkstra's Algorithm

Dijkstra's algorithm clearly cannot work when a path cost is more than `INT_MAX`. A reasonable-looking restriction is to bound edge costs by $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}-1} \right\rfloor$, since the longest optimal path has $\texttt{size} - 1$ links and so the most expensive possible path costs no more than `INT_MAX`. However, this has two flaws.

First, since we are writing real code in C, rather than pseudocode in an idealized setting, we must reserve some concrete `int` value `inf` for "infinity". Suppose we set $\texttt{inf} = \texttt{INT\_MAX}$, and that $\texttt{size} - 1$ divides `INT_MAX`. Now the longest path can have cost $(\texttt{size} - 1) \cdot \left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}-1} \right\rfloor = \texttt{INT\_MAX} = \texttt{inf}$. This creates an unpleasant ambiguity: we cannot tell if the farthest vertex is unreachable, or if it is reachable with legitimate cost `INT_MAX`. We need to adjust our maximum edge weights to leave room for `inf`; using $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor$ solves this first issue.

Second, even though the best-known distances start at `inf` (see line 8) and only ever decrease from there, the code can overflow on lines 17 and 18. Consider applying Dijkstra's algorithm on a 32-bit unsigned machine to the graph in Fig. 2. The `size` of the graph is 3 nodes, and the proposed edge-weight upper bound is $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor = \left\lfloor \frac{(2^{32}-1)-1}{3-1} \right\rfloor = 2^{31} - 1$, for example as in the graph pictured in Fig. 2. A glance at the figure shows that the true distance from the source A to vertices B and C are $2^{31} - 1$ and $2^{32} - 2$ respectively. Both values are representable with 32 bits, and neither distance is $\texttt{inf} = 2^{32} - 1$, so naïvely all seems well. Unfortunately, Dijkstra's algorithm does not exactly work like that.

After processing vertices A and B, $2^{31} - 1$ and $2^{32} - 2$ *are* the costs reflected in the `dist` array for B and C respectively—*but unfortunately vertex C is still in the priority queue*. After vertex C is popped on line 12, we fetch its neighbors in the `for` loop; the cost from C to B ($2^{31} - 1$) is fetched on line 15. On line 17 the currently optimal cost to B ($2^{31} - 1$) is compared with the sum of the optimal cost to C ($2^{32} - 2$) plus the just-retrieved cost of the edge from C to B ($2^{31} - 1$). Naïvely, $(2^{32} - 2) + (2^{31} - 1)$ is *greater than* the currently optimal cost $2^{31} - 1$, so the algorithm should stick with the latter. However, $(2^{32} - 2) + (2^{31} - 1)$ overflows, with $\big((2^{32} - 2) + (2^{31} - 1)\big) \texttt{ mod } 2^{32} = 2^{31} - 3$, which is *less than*

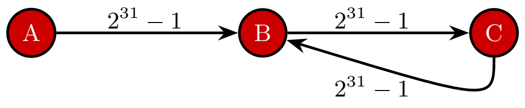


**Fig. 2.** A graph that will result in overflow on a 32-bit machine.

$2^{31}-1$! Thus the code decides that a new cheaper path from A to B exists (in particular, A⇝B⇝C⇝B) and then trashes the `dist` and `prev` arrays on line 18.

Our code uses signed `int` rather than `unsigned int` so we have undefined behavior rather than defined-but-wrong behavior, but the essence of the overflow is identical. We ensure that the "probing edge" does not overflow by restricting the maximum edge cost further, from $\left\lfloor \frac{\texttt{INT\_MAX}-1}{\texttt{size}-1} \right\rfloor$ to $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor$. In Fig. 2, edge weights should be bounded by $\left\lfloor \frac{2^{32}-1}{3} \right\rfloor = 1{,}431{,}655{,}765$; call this value $w$. Suppose we change the edge weights in Fig. 2 from $2^{31}-1$ to $w$. Now vertex B has distance $w$ and C has distance $2 \cdot w$. When we remove C from the priority queue, the comparison on line 17 is between the known best cost to B (*i.e.*, $w$) and the candidate best cost to B via C (*i.e.*, $3 \cdot w = 2^{32}-1 = \texttt{INT\_MAX}$). There is no overflow, so the candidate is rejected and the code behaves as advertised.

We fold these new restrictions into the mathematical graph $\gamma$. In addition to the bounds discussed above, we require a few other more straightforward bounds: edge costs be non-negative, as is typical for Dijkstra; $4 \cdot \texttt{size} \leq \texttt{INT\_MAX}$, to ensure that the multiplication in the `malloc` on line 5 does not overflow; and that $\left\lfloor \frac{\texttt{INT\_MAX}}{\texttt{size}} \right\rfloor \cdot (\texttt{size}-1) < \texttt{inf}$, so no valid path has cost `inf`. These bounds are optimal: if the input is any less restricted, the postcondition will fail. The last restriction on `inf` is not sufficient when $\texttt{size} = 1$, so in that special case we further require that any (self-loop) edges cost less than `inf`. Whenever $0 < 4 \cdot \texttt{size} \leq \texttt{INT\_MAX}$, the restrictions on `inf` are satisfiable with $\texttt{inf} \overset{\Delta}{=} \texttt{INT\_MAX}$.

### 3.3 Related Work on Dijkstra in Algorithms and Formal Methods

We were not able to find a reference that gives a robust, precise, and full description of the overflow issue we describe above. Dijkstra's original paper [22] ignores the issue, as do the standard textbooks *Introduction to Algorithms* (*a.k.a.* CLRS) by Cormen *et al.* [20] and *Algorithm Design* by Kleinberg and Tardos [38]. Sedgewick's book on graph algorithms in C [66] sidesteps the overflow in line 17 by requiring weights be in `double`, which *does* have a well-defined positive infinity value and cannot overflow in the traditional sense; Sedgewick and Wayne's *Algorithms* textbook in Java does the same [67]. However, Sedgewick's sidestep entails enduring the inevitable round-off intrinsic to floating-point arithmetic, and so his algorithm computes approximate optimal costs rather than exact ones. Sedgewick does not specify any bounds on input edge weights, and accordingly does not (and cannot) provide any bound on this accumulated error. Sedgewick is silent on how to handle an `int`-weighted input graph. Skiena's *Algorithm*

*Design Manual* [68] contains code with exactly the bug we identify: he uses integer weights and does not specify any bounds. To its credit, Heineman *et al.*'s *Algorithms in a Nutshell* [33] takes `int` edge weights as inputs and mentions overflow as a possibility. Heineman *et al.* hustle their way around this overflow by performing the arithmetic in line 17 in `long`. However, this cast does not really handle the problem in a fundamental way: if edge weights are given in `long` rather than `int`, then it would be necessary to cast to `long long`; if edge weights are given in `long long`, then Heineman's hustle breaks as there is no bigger type to which to cast. Moreover, Heineman *et al.* do not bound edge weights, so when the cumulative edge weights are too high their code fails silently.

Chen verified Dijkstra in Mizar [15], Gordon *et al.* formalized the reachability property in HOL [29], Moore and Zhang verified it in ACL2 [53], Mange and Kuhn verified it in Jahob [52], Filliâtre in Why3 [25], and Klasen verified it in KeY [37]. Liu *et al.* took an alternative SMT-based approach to verify a Java implementation of Dijkstra [51]. The most recent effort (2019) is by Lammich *et al.*, working within Isabelle/HOL, although they only return the weight of the shortest path rather than the path itself [45]. In general the previous mechanized proofs on Dijkstra verify code defined within idealized formal environments, *e.g.* with unbounded integers rather than machine `int`s and a distinguished non-integer value for infinity. No previous work mentions the overflow we uncover.

## 4 Minimum Spanning Trees

Here we discuss our verifications of the classic MST algorithms Prim and Kruskal. Although our machine-checked proofs are about real C code, in this section we take a higher-level approach than we did in §3, focusing on our key algorithmic findings and overall experience. Accordingly, we only provide pseudocode for Prim's algorithm rather than a decorated program and do not show any code for Kruskal's. Our development contains our C code and formal proofs [1].

### 4.1 Prim's Algorithm

We put the pseudocode for Prim's algorithm in Fig. 3; the code on the left-hand side is directly from CLRS, whereas the code on the right omits line 5 and will be discussed in §4.2. Note that line 12 contains an implicit call to the PQ's `edit_priority`. Since the pseudocode only compares `keys` (*i.e.*, edge weights) rather than doing arithmetic on them *à la* Dijkstra, there are no potential overflows and it is reasonable to set `INF` to `INT_MAX` in C.

Indeed, our initial verifications of C code were largely "turning the crank" once we had the definitions and associated lemma support for pure/abstract undirected graphs, forests, *etc.* discussed in §2.3. Accordingly, our initial contribution was a demonstration that this new graph machinery was sufficient to verify real code. We also proved that our extensions to CertiGraph from §2 were generic rather than verification-specific by reusing much pure and spatial reasoning that had been originally developed for our verification of Dijkstra.

```
1  MST-PRIM(G,w,r):
2   for each u in G.V
3    u.key = INF
4    u.parent = NIL
5   r.key = 0
6   Q = G.V
7   while Q ≠ ∅
8    u = EXTRACT-MIN(Q)
9    for each v in G.Adj[u]
10    if v ∈ Q and w(u,v) < v.key
11     v.parent = u
12     v.key = w(u,v)
```

```
MST-NOROOT-PRIM(G,w):
 for each u in G.V
  u.key = INF
  u.parent = NIL

 Q = G.V
 while Q ≠ ∅
  u = EXTRACT-MIN(Q)
  for each v in G.Adj[u]
   if v ∈ Q and w(u,v) < v.key
    v.parent = u
    v.key = w(u,v)
```

**Fig. 3.** Left: Prim's algorithm from CLRS [20]. Right: the same omitting line 5.

## 4.2 Prim's Algorithm Handles Multiple Components Out of the Box

Textbook discussions of Prim's algorithm are usually limited to single-component input graphs (*a.k.a.* connected graphs), producing a minimum spanning tree. It is widely believed that Prim's is not directly applicable to graphs with multiple components, which should produce a minimum spanning forest. For example, both Rozen [65] and Sedgewick *et al.* [66,67] leave the extension to multiple components as a formal exercise for the reader, whereas Kepner and Gilbert suggest that multiple-component graphs should be handled by first finding the components and then running Prim on each component [36].

After we completed our initial verification, a close examination of our formal invariants showed us that the algorithm *exactly as given by standard textbooks* will properly handle multi-component graphs *in a single run*. The confusion starts because, in a fully connected graph, any vertex `u` removed from the PQ on line 8 must have `u.key` $<$ `INF`; *i.e.*, `u` must be immediately reachable from the spanning tree that is in the process of being built. However, nothing in the code relies upon this connectedness fact! All we need is that `u` is the "closest vertex" to the "current component." If `u.key` $=$ `INF` *and* `u` is a minimum of the PQ, then it simply means that the "previous component" is done, and we have started spanning tree construction on a new unconnected component "rooted" at `u`, yielding a forest. The node `u`'s parent will remain `NIL`, at it was after the setup loop on line 4, indicating that it is the root of a spanning tree. Its `key` will be `INF` rather than 0, but the `keys` are *internal to Prim's algorithm*: clients only get back the spanning forest as encoded in the `parent` pointers[4].

Having made this discovery, we updated our proofs to support the new weaker precondition, which is what we currently formally verify in Coq [71]. A little further thought led to the realization that since Prim can handle arbitrary numbers

[4] The `keys` simply record the edge-weight connecting a vertex to its candidate parent; recall that line 12 is really a call to the PQ's `edit_priority`. If a client wishes to know this edge weight, it can simply look up the edge in the graph.

of components, the initialization of the root's `key` in line 5 is in fact unnecessary. Accordingly, if we remove this line and the associated function argument `r` from `MST-PRIM` (*i.e.*, the code on the right half of Fig. 3), the algorithm still works correctly. Moreover, *the program invariants become simpler* because we no longer need to treat a specified vertex (`r`) in a distinguished manner. Our formal development verifies this version of the algorithm as well [1].

## 4.3 Related Work on Prim in Algorithms and Formal Methods

Prim's algorithm was in fact first developed by the Czech mathematician Vojtěch Jarník in 1930 [35] before being rediscovered by Robert Prim in 1957 [61] and a third time by Edsger W. Dijkstra in 1959 [22]. Both Prim's and Dijkstra's treatment explicitly assumes a connected graph; although we cannot read Czech, some time with Google translate suggests that Jarník's treatment probably does the same. The textbooks we surveyed [20,36,38,65–68] seem to derive from Prim's and/or Dijkstra's treatment. More casual references such as Wikipedia [3] and innumerable lecture slides are presumably derived from the textbooks cited. We have not found any references that state that Prim's algorithm *without modification* applies to multi-component graphs, even when executable code is provided: *e.g.*, Heineman *et al.* provide C++ code that aligns closely with our C code [33], but do not mention that their code works equally well on multi-component graphs. Sadly, many sources promulgate the false proposition that modifications to the algorithm are needed to handle multi-component graphs (*e.g.*, [3,36,65–67]). Likewise, we have found no reference that removes the initialization step (line 5 in Fig. 3) from the standard algorithm.

Prim's algorithm has been the focus of a few previous formalization efforts. Guttman formalised and proved the correctness of Prim's algorithm using Stone-Kleene relation algebras in Isabelle/HOL [30]. He works in an idealized formal environment that does not require the development of explicit data structures; his code does not appear to be executable. Lammich *et al.* provided a verification of Prim's algorithm [45]. Lammich *et al.* also work within the idealized formal environment of Isabelle/HOL, but, in contrast to Guttman, develop efficient purely functional data structures and extract them to executable code. Both Guttman and Lammich explicitly require that the input graph be connected.

## 4.4 Kruskal's Algorithm

Although Kruskal's algorithm is sometimes presented as taking connected graphs and producing spanning trees, the literature also discusses the more general case of multi-component input graphs and spanning forests. However, Kruskal has only recently been the focus of formal verification efforts, partly because it relies on the notoriously difficult-to-verify union-find algorithm; fortunately, the CertiGraph project has an existing fully-verified union-find implementation that we can leverage [73]. Kruskal also requires a sorting function; we implemented `heapsort` as explained in §5.2. Kruskal is optimized for compact representations of sparse graphs, so the $O(1)$ space cost of `heapsort` is a reasonable fit.

The primary interest of our verification of Kruskal is in our proof engineering. Kruskal inputs graphs as edge lists rather than adjacency matrices. In addition to requiring an addition to our spatial graph predicate menu, this means that Kruskal's input graphs can have multiple edges between a given pair of vertices (*i.e.*, a "multigraph"). Pleasingly, we can reuse most of the undirected graph definitions (§2.3), demonstrating that they are generic and reusable.

Another challenge is integrating the pre-existing CertiGraph verification of union-find. We are pleased to say that no change was required for CertiGraph's existing union-find definitions, lemmas, specifications and verification. Kruskal actually manipulates two graphs simultaneously: a directed graph with vertex labels (to store parent pointers and ranks) within union-find, and an undirected multigraph with edge labels (for which the algorithm is constructing a spanning forest). Beyond showing that CertiGraph was capable of this kind of systems-integration challenge, we had to develop additional lemma support to bridge the directed notion of "reachability," used within the directed union-find graph to the undirected notion of "connectedness," used in the MSF graph (§2.3).

### 4.5 Related Work on Kruskal in Algorithms and Formal Methods

Joseph Kruskal published his algorithm in 1956 [42] and it has appeared in numerous textbooks since (*e.g.*, [20,38,66,68]). Kruskal's algorithm is usually preferred over Prim's for sparse graphs, and is sometimes presented as "the right choice" when confronted with multi-component graphs under the mistaken assumption that Prim's first requires a component-finding initial step.

Guttman generalized minimum spanning tree algorithms using Stone relation algebras [31], and provided a proof of Kruskal's algorithm formatted in said algebras. Like in his work on Prim's [30], Guttmann works within Isabelle/HOL and does not include concrete data structures such as priority-queues and union-find, instead capturing their action as equivalence relations in the underlying algebras. In Guttmann's Kruskal paper, he mentions that his Prim paper axiomatizes the fact that "every finite graph has a minimum spanning forest," which he is then able to prove *using his Kruskal algorithm*. Interestingly, our Prim verification needs the same fact, but we prove it directly.

In a similar vein, Haslbeck *et al.* verified Kruskal's algorithm [32] by building on Lammich *et al.*'s earlier work on Prim [45]. Like Lammich *et al.*, Haslbeck *et al.* work within Isabelle/HOL with a focus on purely functional data structures.

One of the stumbling blocks in verifying Kruskal's algorithm is the need to verify union-find. In addition to CertiGraph [73], two recent efforts to certify union-find are by Charguéraud and Pottier, who also prove time complexity [14]; and by Filliâtre [26], whose proof benefits from a high degree of automation.

## 5 Verified Binary Heaps in C

A binary heap embeds a heap-ordered tree in an array and uses arithmetic on indices to navigate between a parent and its left and right children [20]. In addition to providing the standard `insert` and `remove-min`/`remove-max` operations (depending on whether it is a min- or max-ordered heap) in logarithmic time, binary heaps can by upgraded to support two nontrivial operations. First, Floyd's `heapify` function builds a binary heap from an unordered array in linear time, and as a related upgrade, `heapsort` performs a worst-case linearithmic-time sort using only constant additional space. Second, binary heaps can be upgraded to support logarithmic-time `decrease-` and `increase-priority` operations, which we generalize straightforwardly into `edit_priority`.

Binary heaps are a good fit for our graph algorithms because Dijkstra's and Prim's algorithms need to edit priorities, and a constant-space `heapsort` is appropriate for the sparse edge-list-represented graphs typically targeted by Kruskal's. The C language has poor support for polymorphic higher-order functions, and a binary heap that supports `edit_priority` is half as fast as a binary heap that does not. Accordingly, we implement binary heaps in C three times:

| Name | Heap order | `edit_priority` | `heapify` | Payload |
|---|---|---|---|---|
| basic | min | no | yes | `void*` |
| advanced | min | yes | no | `int` |
| Kruskal | max | no | yes | `int, int` (*i.e.*, unboxed) |

Priorities are of type `int`. The Kruskal-specific implementation is stripped down to the bare minimum required to implement `heapsort` (*e.g.*, it does not support `insert`). We next overview these verifications in three parts: basic heap operations, `heapify` and `heapsort` operations, and the `edit_priority` operation.

## 5.1 The Basic Heap Operations of Insertion and Min/Max-Removal

Because we are juggling three implementations, we take some care to factor our verification to maximize reuse. First, each C implementation has its own exchange and comparison functions that handle the nitty-gritty of the payload and choose between a min or max heap. The following lines are from the "basic" implementation, in which the "payload" (`data` field) is of type `void*`:

```
void exch(unsigned int j, unsigned int k, Item arr[]) {
 int priority = arr[j].priority; void* data = arr[j].data;
 arr[j].priority = arr[k].priority; arr[j].data = arr[k].data;
 arr[k].priority = priority; arr[k].data = data; }
int less(unsigned int j, unsigned int k, Item arr[]) {
 return (arr[j].priority <= arr[k].priority); }
```

These C functions are specified as refinements of Gallina functions that exchange polymorphic data in lists and compare objects in an abstract preordered set; we verify them in VST after a little irksome engineering. The payoff is that the key heap operations, which, following Sedgewick [66], we call `swim` and `sink`, can use identical C code (up to alpha renaming) in all three implementations:

```
void swim(unsigned int k, Item arr[]) {
 while (k > ROOT_IDX && less (k, PARENT(k), arr)) {
  exch(k, PARENT(k), arr); k = PARENT(k);        } }
void sink (unsigned int k, Item arr[], unsigned int available) {
```

```
15  while (LEFT_CHILD(k) < available) {
16   unsigned j = LEFT_CHILD(k);
17   if (j+1 < available && less(j+1, j, arr)) j++;
18   if (less(k, j, arr)) break; exch(k, j, arr); k = j;     } }
```

These functions involve a number of complexities, both at the algorithms level and at the semantics-of-C level. At the C level, there is the potential for a rather subtle bug in the macros ROOT_IDX, PARENT, etc. Abstractly, these are simple: the root is in index 0; the children of $x$ at roughly $2x$ and the parent at roughly $\frac{x}{2}$, with $\pm 1$ as necessary. The danger is thinking that because the variables are `unsigned int`, all arithmetic will occur in this domain; in fact we must force the associated constants into `unsigned int` as well:

```
#define ROOT_IDX  0u
#define PARENT(x) (x-1u)/2u
#define LEFT_CHILD(x) (2u*x)+1u
#define RIGHT_CHILD(x) 2u*(x+1u)
```

A second C-semantics issue is the potential for overflow within LEFT_CHILD and RIGHT_CHILD (as well as the increments on line 17), and underflow within the PARENT macro (if x should ever be 0). To avoid this overflow, the precondition of `sink` requires that when `k` is in bounds (*i.e.*, $\texttt{k} < \texttt{available}$), then $2 \cdot (\texttt{available} - 1) \leq \texttt{max\_unsigned}$. An edge case occurs when deleting the last element from a heap ($\texttt{k} = \texttt{available}$); we then require $2 \cdot \texttt{k} \leq \texttt{max\_unsigned}$.

At the algorithmic level, both the `swim` and `sink` functions involve nontrivial loop invariants; `sink` is complicated by the further need to support Floyd's `heapify`, during which a large portion of the array is unordered. Accordingly, we build Gallina models of both functions and show that they restore heap order given a mostly-ordered input heap. There are two different versions of "mostly-ordered". Specifically, `swim` uses a "bottom-up" version:

```
Definition weak_heapOrdered2 (L : list A) (j : nat) : Prop :=
 (∀ i b, i ≠ j → nth_error L i = Some b →
   ∀ a, nth_error L (parent i) = Some a → a ≼ b) ∧
 (grandsOk L j root_idx).
```

whereas `sink` uses a "top-down" version:

```
Definition weak_heapOrdered_bounded (L:list A) (k:nat) (j:nat) :=
 (∀ i a, i ≥ k → i ≠ j → nth_error L i = Some a →
  (∀ b, nth_error L (left_child i) = Some b → a ≼ b) ∧
  (∀ c, nth_error L (right_child i) = Some c → a ≼ c)) ∧
 (grandsOk L j k).
```

The parameter `j` indicates a "hole", at which the heap may not be heap-ordered; `grandsOk` bridges this hole by ordering the parent and the children of `j`:

```
Definition grandsOk (L : list A) (j : nat) (k : nat) : Prop :=
  j ≠ root_idx → parent j ≥ k →
    ∀ gs bb, parent gs = j → nth_error L gs = Some bb →
      ∀ a, nth_error L (parent j) = Some a → a ≼ bb.
```

The parameter `k` is used to support Floyd's `heapify`: it bounds the portion of the list in which elements are heap-ordered (with the exception of `j`). The proofs that the Gallina `swim` and `sink` can restore (bounded) heap-orderedness involve a number of edge cases, but given the above definitions go through. The invariants of the C versions of `swim` and `sink` are stated via the associated Gallina versions, thereby delegating all heap-ordering proofs to the Gallina versions.

The insertion and remove functions we verify are in fact "non-checking" versions (`insert_nc` and `remove_nc`): their preconditions assume there is room in the heap to add or an item in the heap to remove. In the context of Dijkstra and Prim, these preconditions can be proven to hold. The associated verifications involve a little separation logic hackery (specifically, to FRAME away the "junk" part of the heap-array from the "live" part), but are straightforward using VST. We avoid the overflow issue in `sink` by bounding the maximum capacity of the heap: $4 \leq 12 \cdot \texttt{capacity} \leq \texttt{max\_unsigned}$; the magic number 12 comes from the size of the underlying data structure in C. We require users to prove this bound on heap creation, and thereafter handle it under the hood.

## 5.2 Bottom-Up Heapify and Heapsort

Floyd's bottom-up procedure for constructing a binary heap in linear time, and using a binary heap to sort, are classics of the literature [20,66]. Happily, while the asymptotic bound on heap construction is nontrivial, the implementations of both are basically repeated calls to `sink` (and exchanges to remove the root):

```
19 void build_heap(Item arr[], unsigned int size) {
20  unsigned int start = PARENT(size);
21  while(1) { sink(start, arr, size);
22             if (start == 0) break; start--;  } }
23 void heapsort_rev(Item* arr, unsigned int size) {
24  build_heap(arr,size);
25  while (size > 1) { size--;
26   exch(ROOT_IDX, size, arr); sink(ROOT_IDX, arr, size); } }
```

Given that in §5.1 we already generalized the specification for `sink` to handle a portion of the array being unordered, the verification of these functions is straightforward. There is, however, the possibility of a subtle underflow on line 20, in the case when building an empty heap (*i.e.*, `size` = 0). In turn, this means that `heapsort_rev` as given above cannot sort empty lists; in our "basic" implementation we strengthen the precondition accordingly, whereas in our "Kruskal" implementation we add a line before 24 that `returns` when `size` = 0. We use a max-heap for Kruskal because heapsort yields a *reverse* sorted list.

## 5.3 Modifying an Element's Priority

To support edit-priority, each live item is associated not only with its usual `int` priority but also given a unique `unsigned int` "key", generated during `insert` and returned to the client. The binary heap internally maintains a secondary array `key_table` that maps each key to the current location of the associated item within the primary heap array. The client calls `edit_priority` by supplying the key for the item that it wishes to modify, which the binary heap looks up in the `key_table` to locate the item in the heap array before calling `sink` or `swim`. To keep everything linked together, the `key_table` is modified during `exchange`.

To generate the keys on insert, we store a key field within each heap-item in the main array. These keys are initialized to 0..(`capacity` − 1), and thereafter are never modified other than when two cells are swapped during `exchange`. An invariant can then be maintained that the keys from the "live" and "junk" parts have no duplicates. On insertion, we "recycle" the key of the first "junk" item, which is by the invariant known to be appropriately fresh.

### 5.4 Related Work on Binary Heaps in Algorithms and Formal Methods

J. W. J. Williams published the binary heap data structure, along with heapsort, in June 1964 [28]. Floyd proposed his linear-time bottom-up method to construct such heaps that December [27]. Since then, binary heaps, including Floyd's construction and heapsort, have become a staple of the introductory data structure diet [20]. On the other hand, standard textbooks are surprisingly vague on the implementation of `edit_priority` [20,38,66], and completely silent on the generation of fresh keys during insertion. Our method above of "recycling keys" avoids a subtle overflow in a naïve approach, and does not appear in the literature we examined. The naïve idea is to have a global counter starting at 0, which is then increased on each insert. Unfortunately, this is unsound: during (very) long runs involving both `insert` and `remove-min`, this key counter will overflow. Although overflow is defined in C for `unsigned int`, this overflow is fatal algorithmically: multiple live items could be assigned the same key.

Binary heaps have been verified several times in the literature. They were problem 2 of the VACID-0 benchmark [49], and solved in this regard as well by the Why3 team [69]. These solutions did not implement bottom-up heap construction or edit priority. Summers verified heapsort in Viper, again without bottom-up heap construction [56]. Lammich verified Introsort, which includes a heapsort subroutine [44]. Previous formal work ignores nitty-gritty C issues such as the difference between signed and unsigned arithmetic. We believe we are the first formally verified binary heap to support edit-priority.

## 6 Engineering Considerations

Verifying real code is meaningfully harder than verifying toy implementations. On top of such challenges, verifying graph algorithms requires a significant amount of mathematical machinery: there are many plausible ways to define basic notions such as reachability, but not all of them can handle the challenges of verifying real code [72]. Moreover, we would like our mathematical, spatial, and verification machinery to be generic and reusable.

All of the above suggests that it is important to work within existing formal proof developments due a strong desire to not reinvent very large wheels (the existing proof bases we work with contain hundreds of thousands of lines of formal proof). We chose to work with the CompCert certified compiler [50]; the Verified Software Toolchain [4], which provides significant tactic support for separation logic-based deductive verification of CompCert C programs; and the CertiGraph framework [73], which provides much pure and spatial reasoning support for verifying graph-manipulating programs within VST. We did so because these frameworks can handle the challenges of real code and because the CertiGraph included several fully verified implementations of union-find that we wished to reuse in our verification of Kruskal's algorithm.

Modular formal proof development involves major software engineering challenges [64]. Accordingly, we took care factoring our extensions to CertiGraph into generic and reusable pieces. This factoring allows us to reuse machinery between verifications, including in the mathematical, spatial, and verification levels. So, *e.g.*, we share significant pure and spatial machinery between Dijkstra, Prim, and Kruskal. Moreover, we maintain good separation between pure and spatial reasoning. So, *e.g.*, both our Dijkstra and Prim verifications can handle multiple spatial variants of adjacency matrices without significant change.

On the other hand, working within existing developments involves some challenges, primarily in that some design decisions have been already made and are hard to change. Moreover, our verifications tickled numerous bugs within VST, including: overly-aggressive automatic entailment simplifying, poor error messages, improper handling of C `structs`, and performance issues. We have been fortunate that the VST team has been willing to work with us to fix such bugs, although some work still remains. Performance remains one area of focus: for example, checking our verification of Kruskal with a 3.7 GHz processor and 32 gb of memory takes more than 22 min even after all of the generic pure and spatial reasoning has been checked, *i.e.* approximately 7 s per line of C code (including whitespace and comments). This performance is unviable for verifying an industrial-sized application of equivalent difficulty: *e.g.*, it would take 13 years for Coq to check the proof for 1,000,000 lines of C. Before some optimizations to our proof structure, the time was significantly longer still.

Our contributions to CertiGraph include pieces that are reused repeatedly and pieces that are more bespoke. Below, we give a sense of both the size of our development (lines of formal Coq proof) and the mileage we get out of our own work via reuse. Items "added with +" are very similar (within 1%) of each other; Prim #4 is the version that does not set the root, *i.e.* on the right in Fig. 3.

| Name | Used | LoC | Name | LoC |
|---|---|---|---|---|
| MathAdjMat | 7x | 165 | DijkSpec1+2+3 | 301 |
| Undirected | 5x | 2,139 | VerifDijk1+2+3 | 3,554 |
| MathUAdjMat | 4x | 1,024 | PrimSpec1+2+3+4 | 508 |
| SpaceAdjMat1+2+3 | 7x | 499 | VerifPrim1+2+3+4 | 7,455 |
| EdgeListGraph | 1x | 911 | KruskalSpec | 302 |
| MathDijkGraph | 3x | 165 | VerifKruskal | 1,606 |
| DijkPureProof | 3x | 2,124 | VerifHeapSort | 568 |
| UndirectedUF | 1x | 183 | VerifBasicBinaryHeap | 777 |
| BinaryHeapModel | 1x | 1,870 | VerifAdvBinaryHeap | 2,253 |
| Total (pure/spatial) | | 9,080 | Total (verifications) | 17,234 |

In total we have 26,314 novel lines of Coq proof to verify 1,155 lines of C code divided among 12 files, including 3 variants of Dijkstra, 4 variants of Prim, 1 of Kruskal (which includes its `heapsort`), and 2 binary heaps.

## 7 Concluding Thoughts: Related and Future Work

We have already discussed work directly related to Dijkstra's (§3.3), Prim's (§4.3), and Kruskal's (§4.5) algorithms, as well as binary heaps (§5.4). Summarizing briefly to the point of unreasonableness, our observations about Dijkstra's overflow and Prim's specification are novel, and existing formal proofs focus on code working within idealized environments rather than handling the real-world considerations that we do. We have also discussed the three formal developments we build upon and extend: CompCert, VST, and CertiGraph (Sect. 6). Our goal now is to discuss mechanized graph reasoning and verification more broadly.

*Reasoning About Mathematical Graphs.* There is a 30+ year history of mechanizing graph theory, beginning at least with Wong [74] and Chou [19] and continuing to the present day; Wang discusses many such efforts [72, §3.3]. The two abstract frameworks that seem closest to ours are those by Noschinski [58]; and by Lammich and Nipkow [45]. The latter is particularly related to our work, because they too start with a directed graph library and must extend it to handle undirected graphs so that they can verify Prim's algorithm.

*More-Automated Verification.* Broadly speaking, mechanized verification of software falls in a spectrum between more-automated-but-less-precise verifications and less-automated-but-more-precise verifications. Although VST contains some automation, we fall within the latter camp. In the former camp, landmark initial separation logic [63] tools such as Smallfoot [7] have grown into Facebook's industrial-strength Infer [11]. Other notable relatively-automated separation logic-based tools include HIP/SLEEK [17], Bedrock [18], KIV [24], VerCors [9], and Viper [57]. More-automated solutions that use techniques other than separation logic include Boogie [6], Blast [8], Dafny [48], and KeY [2]. In Sect. 3.3 we discuss how some of these more-automated approaches have been applied to verify Dijkstra's algorithm. Petrank and Hawblitzel's Boogie-based verification of a garbage collector [60], Bubel's KeY-based verification of the Schorr-Waite algorithm, and Chen *et al.*'s Tarjan's strongly connected components algorithm in (among others) Why3 [16] are three examples of more-automated verification of graph algorithms. Müller verified *binomial* (not binary) heaps in Viper, although his implementation did not support an edit-priority function [55]. The VOCAL project has verified a number of data structures, including binary and other heaps (all without edit-priority) and union-find [13].

We are not confident that more-automated tools would be able to replicate our work easily. We prove full functional correctness, whereas many more-automated tools prove only more limited properties. Moreover, our full functional correctness results rely upon a meaningful amount of domain-specific knowledge about graphs, which automated tools usually lack. Even if we restrict ourselves to more limited domains such as overflows, several more automated efforts did not uncover the overflow that we described in Sect. 3.3. The proof that certain bounds on edge weights and `inf` suffice depends on an intimate understanding of Dijkstra's algorithm (in particular, that it explores one edge beyond the optimum paths); overall the problem seems challenging in highly-automated settings. The more powerful specification we discover for Prim's algorithm in Sect. 4.2 is likewise not something a tool is likely to discover: human insight appears necessary, at least given the current state of machine learning techniques.

In contrast, several of the potential overflows in our binary heap might be uncovered by more-automated approaches, especially those related to the `PARENT` and `LEFT_CHILD` macros from Sect. 5.1. Although the arithmetic involves both addition/subtraction and multiplication/division, we suspect a tool such as Z3 [54] could handle it. Moreover, a sufficiently-precise tool would probably spot the necessity of forcing the internal constants into `unsigned int`. The issue of sound key generation described in Sect. 5.3 might be a bit trickier. On the one hand, `unsigned int` overflow is defined in C, so real code sometimes relies upon it. Accordingly, merely observing that the counter could overflow does not guarantee that the code is necessarily buggy. On the other hand, some tools might flag it anyway out of caution (*i.e.* right answer, wrong reason).

*Less-Automated Verification.* Although as discussed above some more-automated tools have been applied to verify graph algorithms, the problem domain is sufficiently complex that many of the verifications discussed in Sect. 3.3, Sect. 4.3, and Sect. 4.5 use less-automated techniques. Two basic approaches are popular. The "shallow embedding" approach is to write the algorithm in the native language of a proof assistant. The "deep embedding" approach is to write the algorithm in another language whose semantics has been precisely defined in the proof assistant. VST uses a deep embedding, and so we do too; one of VST's more popular competitors in the deep embedding style is "Iris Proof Mode" [39]. In contrast, Lammich *et al.* have produced a series of results verifying a variety of graph algorithms using a shallow embedding (*e.g.*, [32,43,45–47]). From a bird's-eye view Lammich *et al.*'s work is the most related to our results in this paper: they verify all three algorithms we do and are able to extract fully-executable code, even if sometimes their focus is a bit different, *e.g.* on novel purely-functional data structures such as a priority queue with `edit_priority`.

*Pen-and-Paper Verification of Graph Algorithms.* We use separation logic [63] as our base framework. Initial work on graph algorithms in separation logic was minimal; Bornat *et al.* is an early example [10]. Hobor and Villard developed the technique of ramification to verify graph algorithms [34], using a particular "star/wand" pattern to express heap update. Wang *et al.* later integrated ramification into VST as the CertiGraph project we use [73]. Krishna *et al.* [40] have developed a flow algebraic framework to reason about local and global properties of *flow graphs* in the program heap; their flow algebra is mainly used to tackle local reasoning of global graphs in program heaps. Flow algebras should be compatible with existing separation logics; implementation and integration with the Iris project appears to be work in progress [41].

Krishna *et al.* are interested in concurrency [40]; Raad *et al.* provide another example of pen-and-paper reasoning about concurrent graph algorithms [62].

*Future Work.* We see several opportunities for decreasing the effort and/or increasing the automation in our approach. At the level of Hoare tuples, we see opportunities for improved VST tactics to handle common cases we encounter in graph algorithms. At the level of spatial predicates, we can continue to expand our library of graph constructions, for example for adjacency lists. We also believe there are opportunities to increase modularity and automation at the interface between the spatial and the mathematical levels, *e.g.* we sometimes compare C pointers to heap-represented graph nodes for equality, and due to the nature of our representations this equality check will be well-defined in C when the associated nodes are present in the mathematical graph, so this check should pass automatically.

We believe that more automation is possible at the level of mathematical graphs: for example reachability techniques based on regular expressions over matrices and related semirings [5,23,70]. We are also intrigued by the recent development of various specialized graph logics such as by Costa *et al.* [21] and hope that these kinds of techniques will allow us to simplify our reasoning. The key advantage of having end-to-end machine-checked examples such as the ones we presented above is that they guide the automation efforts by providing precise goals that are known to be strong enough to verify real code.

*Conclusion.* We extend the CertiGraph library to handle undirected graphs and several flavours of graphs with edge labels, both at the pure and at the spatial levels. We verify the full functional correctness of the three classic graph algorithms of Dijkstra, Prim, and Kruskal. We find nontrivial bounds on edge costs and infinity for Dijkstra and provide a novel specification for Prim. We verify a binary heap with Floyd's `heapify` and `edit_priority`. All of our code is in CompCert C and all of our proofs are machine-checked in Coq.

**Acknowledgements.** We thank Shengyi Wang for his help and support.

## References

1. Functional Correctness of C implementations of Dijkstra's, Kruskal's, and Prim's Algorithms (2021). https://doi.org/10.5281/zenodo.4744664
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification-The KeY Book-From Theory to Practice (2016)
3. Anonymous: Prim's algorithm. https://en.wikipedia.org/wiki/Prim%27s_algorithm
4. Appel, A.W., et al.: Program Logics for Certified Compilers. Cambridge University Press, Cambridge (2014)
5. Backhouse, R., Carré, B.: Regular algebra applied to path-finding problems. J. Inst. Math. Appl. **15**, 161–186 (1975)
6. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_6
8. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transf. **9**, 505–525 (2007)
9. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
10. Bornat, R., Calcagno, C., O'Hearn, P.: Local reasoning, separation and aliasing. In: SPACE (2004)
11. Calcagno, C., et al.: Moving fast with software verification. In: NASA Formal Methods Symposium (2015)
12. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: ICFP (2011)
13. Charguéraud, A., Filliâtre, J.C., Pereira, M., Pottier, F.: VOCAL - a verified OCaml library. ML Family Workshop (2017)
14. Charguéraud, A., Pottier, F.: Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. J. Autom. Reason. **62**, 331–365 (2019)
15. Chen, J.C.: Dijkstra's shortest path algorithm. JFM **15**, 237–247 (2003)
16. Chen, R., Cohen, C., Lévy, J., Merz, S., Théry, L.: Formal proofs of Tarjan's strongly connected components algorithm in Why3, Coq and Isabelle. In: ITP (2019)
17. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. Comput. Program. **77**, 1006–1036 (2010)

18. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI (2011)
19. Chou, C.T.: A formal theory of undirected graphs in HOL. In: HOL (1994)
20. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.S.: Introduction to Algorithms, 3rd edn. (2009)
21. Costa, D., Brotherston, J., Pym, D.: Graph decomposition and local reasoning (2020). Under submission
22. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
23. Dolan, S.: Fun with semirings: a functional pearl on the abuse of linear algebra. In: ICFP (2013)
24. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV - overview and VerifyThis competition. STTT **17**, 677–694 (2015)
25. Filliâtre, J.C.: Dijkstra's shortest path algorithm in Why3 (2011). http://toccata.lri.fr/gallery/dijkstra.en.html
26. Fillitre, J.C.: Simpler proofs with decentralized invariants. J. Log. Algebraic Methods Program. **121**, 100645 (2021)
27. Floyd, R.W.: Algorithm 245: treesort. Commun. ACM **7**(12), 701 (1964)
28. Forsythe, G.E.: Algorithms. Commun. ACM **7**(6), 347–349 (1964)
29. Gordon, M., Hurd, J., Slind, K.: Executing the formal semantics of the accellera property specification language by mechanised theorem proving. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 200–215. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_19
30. Guttmann, W.: Relation-algebraic verification of prim's minimum spanning tree algorithm. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 51–68. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_4
31. Guttmann, W.: Verifying minimum spanning tree algorithms with stone relation algebras. J. Log. Algebraic Methods Program. **101**, 132–150 (2018)
32. Haslbeck, M.P.L., Lammich, P.: Refinement with time - refining the run-time of algorithms in Isabelle/HOL. In: ITP (2019)
33. Heineman, G., Pollice, G., Selkow, S.: Algorithms in a Nutshell. O'Reilly (2008)
34. Hobor, A., Villard, J.: Ramifications of sharing in data structures. In: POPL (2013)
35. Jarník, V.: O jistém problému minimálním. (z dopisu panu o. Borůvkovi) (1930)
36. Kepner, Jeremy; Gilbert, J.: Graph algorithms in the language of linear algebra. Soc. Ind. Appl. Math. (2011)
37. Klasen, V.: Verifying Dijkstra's algorithm with KeY. Diploma thesis (2010)
38. Kleinberg, J.M., Tardos, É.: Algorithm Design. Addison-Wesley (2006)
39. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: POPL (2017)
40. Krishna, S., Shasha, D., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. In: POPL (2017)
41. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: ESOP (2020)
42. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Am. Math. Soc. **7**, 48–50 (1956)
43. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 325–340. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_21
44. Lammich, P.: Efficient verified implementation of Introsort and Pdqsort. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 307–323. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_18

45. Lammich, P., Nipkow, T.: Proof pearl: Purely functional, simple and efficient priority search trees and applications to Prim and Dijkstra. In: ITP (2019)
46. Lammich, P., Sefidgar, S.R.: Formalizing the Edmonds-Karp algorithm. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 219–234. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_14
47. Lammich, P., Sefidgar, S.R.: Formalizing network flow algorithms: a refinement approach in Isabelle/HOL. J. Autom. Reason. **62**(2), 261–280 (2019)
48. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
49. Leino, K.R.M., Moskal, M.: VACID-0: verification of ample correctness of invariants of data-structures. Edition 0 (2010)
50. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL (2006)
51. Liu, T., Nagel, M., Taghdiri, M.: Bounded program verification using an SMT solver: a case study. In: ICST (2012)
52. Mange, R., Kuhn, J.: Verifying Dijkstra's algorithm in Jahob (2007)
53. Moore, J.S., Zhang, Q.: Proof Pearl: Dijkstra's shortest path algorithm verified with ACL2. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 373–384. Springer, Heidelberg (2005). https://doi.org/10.1007/11541868_24
54. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
55. Müller, P.: The binomial heap verification challenge in Viper. In: Müller, P., Schaefer, I. (eds.) Principled Software Development. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-98047-8_13
56. Müller, P.: Private correspondence (2021)
57. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
58. Noschinski, L.: A graph library for Isabelle. Math. Comput. Sci. **9**, 23–39 (2015)
59. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44802-0_1
60. Petrank, E., Hawblitzel, C.: Automated verification of practical garbage collectors. Log. Methods Comput. Sci. **6** (2010)
61. Prim, R.C.: Shortest connection networks and some generalizations. Bell Syst. Tech. J. **36**(6), 1389–1401 (1957)
62. Raad, A., Hobor, A., Villard, J., Gardner, P.: Verifying concurrent graph algorithms. In: Igarashi, A. (ed.) APLAS 2016. LNCS, vol. 10017, pp. 314–334. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47958-3_17
63. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS (2002)
64. Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: QED at large: a survey of engineering of formally verified software. CoRR (2020)
65. Rosen, K.H.: Discrete Mathematics and Its Applications. 7th edn. (2012)
66. Sedgewick, R.: Algorithms in C, Part 5: Graph Algorithms (2002)
67. Sedgewick, R., Wayne, K.: Algorithms. 4th edn. Addison-Wesley (2011)

68. Skiena, S.: The Algorithm Design Manual, 2nd edn. Springer, Heidelberg (2008)
69. Tafat, A., Marché, C.: Binary heaps formally verified in Why3 (2011)
70. Tarjan, R.E.: A unified approach to path problems. J. ACM **28**(3), 577–593 (1981)
71. Coq development team: The Coq Proof Assistant. https://coq.inria.fr/
72. Wang, S.: Mechanized verification of graph-manipulating programs. Ph.D. thesis, National University of Singapore (2019)
73. Wang, S., Cao, Q., Mohan, A., Hobor, A.: Certifying graph-manipulating C programs via localizations within data structures. In: OOPSLA (2019)
74. Wong, W.: A simple graph theory and its application in railway signaling. In: HOL Theorem Proving System and Its Applications (1991)

# Gillian, Part II: Real-World Verification for JavaScript and C

Petar Maksimović[1(✉)], Sacha-Élie Ayoun[1], José Fragoso Santos[2], and Philippa Gardner[1]

[1] Imperial College London, London, UK
{p.maksimovic,s.ayoun17,p.gardner}@imperial.ac.uk
[2] INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
jose.fragoso@tecnico.ulisboa.pt

**Abstract.** We introduce verification based on separation logic to Gillian, a multi-language platform for the development of symbolic analysis tools which is parametric on the memory model of the target language. Our work develops a methodology for constructing compositional memory models for Gillian, leading to a unified presentation of the JavaScript and C memory models. We verify the JavaScript and C implementations of the AWS Encryption SDK message header deserialisation module, specifically designing common abstractions used for both verification tasks, and find two bugs in the JavaScript and three bugs in the C implementation.

## 1 Introduction

Separation logic (SL) [25,40] introduced compositional program verification using Hoare reasoning. Current analysis tools based on ideas from SL include: the automatic tool Infer [8,9] used inside Facebook to find lightweight bugs in Java/C/C++/Obj-C programs; the semi-automatic tool Verifast [26], which provides full verification for fragments of C and Java; the semi-automatic tool JaVerT [21], which provides bug-finding and verification for JavaScript (JS) programs; and the Viper architecture [36,35], which provides a verification backend for multiple programming languages, including Java, Rust, and Python. Our goal is to introduce verification based on SL to Gillian [19], a multi-language platform for symbolic analysis, integrating bug-finding and verification in the spirit of JaVerT and targeting many languages in the spirit of Viper.

Gillian currently supports three types of program analysis: symbolic testing, verification and bi-abduction. In [19], the focus was on symbolic testing, parametrised on complete concrete and symbolic memory models of the target language (TL), and underpinned by a core symbolic execution engine with strong mathematical foundations. Gillian analysis is done on GIL, an intermediate goto language parametric on a set of memory actions, which describe the fundamental ways in which TL programs interact with their memories. To instantiate Gillian to a new TL, a tool developer must: (1) identify the set of the TL memory actions and implement the TL memory models using these actions; and (2) provide a trusted compiler from the TL to GIL, which preserves the TL memory models and the semantics. In [19], Gillian was instantiated to JS and C, and used to find bugs in two real-world data-structure libraries, Buckets.js [43] and Collections-C [41]. Here, we introduce compositional memory models for Gillian, extend Gillian analysis with verification based on separation logic, adapt Gillian-JS and Gillian-C to this compositional setting, and provide verified specifications of the JS and C implementations of the deserialisation module of the AWS Encryption SDK.

The compositional Gillian memory models (§2) are given by the tool developer for each TL instantiation. They are based on partial memories, and formulated using core predicates and the associated consumer and producer actions. Core predicates describe fundamental units of TL memories: e.g., a property of a JS object and a C block cell. Consumers and producers, respectively, frame off and frame on the TL memory resource described by the core predicates. Partiality and frame are familiar concepts from SL [25,40,11]. What is perhaps less familiar is our emphasis on negative resource: i.e., the resource known to be absent from the partial memory. For example, in JS, a new extensible object is known not to contain any property; and, in C, a freed block is known not to be in memory and a cell is known not to exist beyond the block bound. We introduce a methodology for designing Gillian compositional memory models, and apply it to JS and C (§3), resulting in a unexpected similarity between the two models. Our compositional JS memory models follow those given in work on a JS program logic [24] and the JaVerT tool [21], where negative resource was essential for frame preservation, inspired by the use of negative resource to capture stability properties in the CAP concurrent separation logic [14], now used in Iris [27]. Our compositional C memory models are based on the complete CompCert memory model [31]. Despite a large body of work on separation logic for C, we were unable to find a partial C memory model that captures the negative resource in its entirety. The nearest is probably the CH20 formalism [29], which handles freed locations but not block bounds. Negative resource for freed locations has also been used in incorrectness logic [39], and for block bounds in a program logic for WebAssembly [48].

We build Gillian verification on top of our compositional memory models. In particular, using the core predicates, we design an assertion language for writing function specifications in separation logic and, using the consumers and producers, we build a fully parametric spatial entailment engine which enables the use of function specifications in symbolic execution. Gillian also supports user-defined predicates, which allow tool developers to identify the TL language interface familiar to code developers, and code developers to describe and prove properties about the particular data structures in their programs.

We extend Gillian-JS and Gillian-C to enable verification, introducing the JS and C compositional memory models, and using the same trusted compilers as in [19]. With these instantiations, we provide functionally-correct, verified specifications of the message header deserialisation module of the AWS Encryption SDK JS and C implementations (§4, §5). This is stable, critical, industry-grade code (~200loc for JS, ~950loc for C), which uses advanced language features to manipulate complex data structures. To verify this code, we create language-independent predicates to capture the message header, which we then connect without modification to both JS and C memories, giving specifications for the module functions. We also build a library of associated lemmas, used for the verification of both implementations. The verification itself required a substantial improvement of the reasoning capabilities of Gillian, especially when it came to handling arrays of symbolic size. We discovered two bugs in the JS implementation: one a form of prototype poisoning, predicted theoretically in our paper on JaVerT [21]; and another that allowed third parties to potentially alter authenticated, non-secret data. We have also discovered three bugs in the C implementation: one which allowed some malformed headers to be parsed as correct; one over-allocation; and one undefined behaviour. All of these bugs have been fixed.

## 2 Gillian Verification

We introduce Gillian verification based on separation logic (§2.2), extending the GIL execution engine presented in [19] with compositional memory models (§2.1).

### 2.1 Compositional Memory Models

GIL is a simple goto intermediate language whose syntax is given below. It is parametric on a set of TL memory actions, $A \ni \alpha$, given per instantiation by the tool developer. GIL values, $v \in \mathcal{V}al$, contain numbers, strings, booleans, uninterpreted symbols (used, e.g., to represent memory locations), simple types (e.g., numbers, strings), function identifiers and lists of values. GIL expressions, $e \in \mathcal{E}xpr$, contain values, program variables, and unary and binary operators (e.g. addition, list concatenation); GIL symbolic expressions, $\hat{e} \in \hat{\mathcal{E}xpr}$, are analogous except that symbolic variables, $\hat{x} \in \hat{\mathcal{X}}$, are used instead of program variables.

**GIL Syntax**

$$\begin{array}{ll}
\multicolumn{2}{l}{v \in \mathcal{V}al \triangleq i, j, n \in \mathcal{N} \mid s \in \mathcal{S} \mid b \in \mathcal{B} \mid \varsigma \in \mathcal{U} \mid \tau \in \mathcal{T} \mid f \in \mathcal{F} \mid \overline{v} \in List(\mathcal{V}al)} \\
e \in \mathcal{E}xpr \triangleq v \mid x \in \mathcal{X} \mid \ominus e \mid e_1 \oplus e_2 & \hat{e} \in \hat{\mathcal{E}xpr} \triangleq v \mid \hat{x} \in \hat{\mathcal{X}} \mid \ominus \hat{e} \mid \hat{e}_1 \oplus \hat{e}_2 \\
c \in \mathcal{C}md \triangleq x := e \mid \mathsf{ifgoto}\ e\ \ i \mid x := e(e') \mid x := \alpha(e) \mid & func \in \mathcal{F}unc \triangleq f(x)\{\overline{c}\} \\
\qquad\qquad x := \mathsf{uSym}/\mathsf{iSym}(e) \mid \mathsf{return}\ e \mid \mathsf{fail}\ e \mid \mathsf{vanish} & \mathsf{p} \in \mathcal{P}rog = \mathcal{P}_!(\mathcal{F}unc)
\end{array}$$

GIL commands, $c \in \mathcal{C}md$, contain variable assignment, conditional goto, function call, memory actions, allocation of uninterpreted/interpreted symbols, function return, error termination and path cutting. A GIL function, $f(x)\{\overline{c}\}$, comprises an identifier $f \in \mathcal{F}$, a formal parameter $x$[3], and a body given by a list of commands $\overline{c}$. A GIL program is a set of GIL functions with unique identifiers.

GIL execution is defined in terms of state models, which are parametric on a value set, $\mathsf{V} \supseteq \mathcal{V}al$, and a set of memory actions, $A$. We distinguish the Boolean value set, $\Pi \subset \mathsf{V}$, and refer to $\pi \in \Pi$ as a context. State models expose an interface consisting of state actions, $A \uplus A_S$, where the actions

[3] The implementation supports multiple parameters.

$$\frac{\begin{array}{c}\textsc{Memory Action - Success}\\ \mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e) \\ \sigma.\mathrm{eval}_e\,(-) \rightsquigarrow \mathsf{v} \qquad \sigma.\alpha(\mathsf{v}) \rightsquigarrow (\sigma', \mathsf{v}')^{\mathcal{S}} \\ \sigma'.\mathrm{setVar}_x\,(\mathsf{v}') \rightsquigarrow \sigma''\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma'', cs, i{+}1\rangle^{\mathtt{S}}} \qquad \frac{\begin{array}{c}\textsc{Memory Action - Error}\\ \mathsf{cmd}(\mathsf{p}, cs, i) = x := \alpha(e) \\ \sigma.\mathrm{eval}_e\,(-) \rightsquigarrow \mathsf{v} \qquad \sigma.\alpha(\mathsf{v}) \rightsquigarrow (\sigma', \mathsf{v}')^{r} \\ r \neq \mathcal{S} \qquad o = (\text{if } r = \mathcal{E} \text{ then } \mathtt{E} \text{ else } \mathtt{M})\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i\rangle \rightsquigarrow \langle \sigma', cs, i\rangle^{o(\mathsf{v}')}}$$

Fig. 1: GIL Execution Semantics: Memory Actions

$A_S = \{\mathrm{setVar}_x\}_{x\in\mathcal{X}} \cup \{\mathrm{setStore}, \mathrm{getStore}\} \cup \{\mathrm{eval}_e\}_{e\in\mathcal{E}xpr} \cup \{\mathrm{assume}, \mathrm{uSym}, \mathrm{iSym}\}$ address store management, expression evaluation, branching, and allocation.

**Definition 1 (State Model).** *A state model, $S(\mathsf{V}, A) \triangleq \langle |S|, ea\rangle$, comprises: a set of states $\sigma = \langle \mu, \rho, \pi\rangle \in |S|$, containing a memory $\mu$, a variable store $\rho$, and a (satisfiable) context $\pi$*[4]*; and an action execution function, $ea : (A \uplus A_S) \to |S| \to \mathsf{V} \rightharpoonup \mathcal{P}(|S| \times \mathsf{V} \times \mathcal{R})$, with the result $r \in \mathcal{R} = \{\mathcal{S}, \mathcal{E}, \mathcal{M}\}$ denoting success, non-correctible error, or missing information error, pretty-printed $\sigma.\alpha(\mathsf{v}) \to \{(\sigma_i, \mathsf{v}_i)^{r_i}|_{i\in I}\}$ for all outcomes and $\sigma.\alpha(\mathsf{v}) \rightsquigarrow (\mu_i, \sigma_i)^{r_i}$ for a specific outcome, with countable $I$. The value set of concrete state models is the set of GIL values, $\mathcal{V}al$*[5]*; the value set of symbolic state models is the set of symbolic expressions, $\hat{\mathcal{E}}xpr$.*

**Definition 2 (GIL Execution Semantics).** *Given a state model $S$, the GIL execution semantics has judgements of the form:*

$$\mathsf{p} \vdash \langle \sigma, cs, i\rangle^{o} \rightsquigarrow_S \langle \sigma', cs', j\rangle^{o'}$$

*with: call stacks, $cs \in \mathcal{C}all_S$; command indexes, $i, j \in \mathbb{N}$; and outcomes, $o \in \mathcal{O}$.*

The GIL execution semantics is standard for a goto language, except that it is parametrised by the memory actions. Call stacks capture function-related control flow, with $\mathsf{cmd}(\mathsf{p}, cs, i)$ denoting the $i$-th command of the currently executing function (cf. [33] for details). Outcomes, $o \in \mathcal{O} \triangleq \mathtt{S} \mid \mathtt{N}(\mathsf{v}) \mid \mathtt{E}(\mathsf{v}) \mid \mathtt{M}(\mathsf{v})$, indicate how the execution is to proceed: $\mathtt{S}$ states that it can continue; $\mathtt{N}(\mathsf{v})$ states that it terminated normally with return value $\mathsf{v}$; and $\mathtt{E}(\mathsf{v})$ and $\mathtt{M}(\mathsf{v})$ state that it failed with either a non-correctible or missing information error described by $\mathsf{v}$. We give the rules for memory action execution in Figure 1; all can be found in [33].

**Compositional Memory Models.** We move from whole-program memory models [19] to compositional memory models by introducing memory core predicates, $\gamma \in \Gamma$, which represent the fundamental units of the TL memory model (e.g., a memory cell). Core predicates take two lists of parameters, in-parameters (or ins), denoted $\mathsf{v}_i$, and out-parameters (or outs), denoted $\mathsf{v}_o$, such that from the ins we can learn the outs. This concept is similar to predicate parameter modes

[4] States also include allocators (cf. [33] for details), elided to limit clutter.

[5] Note that the only satisfiable concrete context is $\mathsf{true}$, meaning that concrete contexts can be elided and concrete states can be viewed as memory-store pairs, $\langle \mu, \rho\rangle$.

of [37] and we use it to implement a parametric spatial entailment engine. An example of a core predicate is the cell assertion, $x \mapsto \mathsf{v}$, which captures a cell in memory at address $x$ having value $\mathsf{v}$. Its in-parameter is $x$, and its out-parameter is $\mathsf{v}$, because, if we know $x$, we can find $\mathsf{v}$ by looking it up in the memory.

With each core predicate $\gamma \in \Gamma$, we associate a consumer and a producer memory action, denoted by $\mathrm{cons}_\gamma$ and $\mathrm{prod}_\gamma$ respectively, to obtain the set of predicate actions $A_\Gamma = \bigcup_{\gamma \in \Gamma}\{\mathrm{cons}_\gamma, \mathrm{prod}_\gamma\}$, whose meaning is discussed shortly.

**Definition 3 (Compositional Memory Model).** *Given value set* $\mathsf{V}$ *and core predicate set* $\Gamma$*, a compositional memory model,* $M(\mathsf{V}, \Gamma) \triangleq \langle |M|, \mathcal{W}f, \underline{ea}_\Gamma \rangle$*, comprises: **(1)** a partial commutative monoid (PCM)*[6]*,* $|M| = (|M|, \bullet, \mathbf{0})$*, where* $\mathbf{0}$ *denotes the (indivisible) empty memory; **(2)** a well-formedness relation,* $\mathcal{W}f \subseteq |M| \times \Pi$*, with* $\mathcal{W}f_\pi(\mu)$ *denoting that memory* $\mu$ *is well-formed in (satisfiable) context* $\pi$*; and **(3)** a predicate action execution function,* $\underline{ea}_\Gamma : A_\Gamma \times |M| \times \mathsf{V} \times \Pi \rightharpoonup \mathcal{P}(|M| \times \mathsf{V} \times \Pi \times \mathcal{R})$*, pretty-printed* $\mu.\alpha(\mathsf{v})_\pi \to \{(\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}|_{i \in I}\}$ *for all outcomes and* $\mu.\alpha(\mathsf{v})_\pi \rightsquigarrow (\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}$ *for a specific outcome, with countable* $I$*. The value set of concrete memory models is the set of GIL values,* $\mathcal{V}al$*; the value set of symbolic memory models is the set of symbolic expressions,* $\hat{\mathcal{E}}xpr$*.*

We discuss the most important properties that the components of compositional memory models must satisfy; a full list is available in [33]. The PCM requirement is well-known from separation logic [40,11]. Well-formedness holds only for satisfiable contexts, and describes the separation of symbolic resource and any further TL-specific well-formedness criteria (cf. §3). It must be monotonic with respect to context strengthening, compatible with the PCM composition, and the empty memory must be well-formed in any satisfiable context. The action execution function, $\mu.\alpha(\mathsf{v})_\pi \to \{(\mu_i, \mathsf{v}_i)^{r_i}_{\pi_i}|_{i \in I}\}$, denotes that, in a memory $\mu$ that is well-formed in the context $\pi$, executing action $\alpha$ with parameter $\mathsf{v}$ yields a countable number of branches characterised by the non-overlapping[7], satisfiable contexts $\pi_i$, each of which implies $\pi$ and makes the corresponding memory $\mu_i$ well-formed, and all of which together cover $\pi$ (i.e., $\pi \Rightarrow \bigvee_{i \in I} \pi_i$). This last property means that memory actions do not drop paths, which is essential for verification.

The intuition behind consumers and producers is that consumers frame off the core predicate resource (CPR), uniquely determined by the core predicate ins, and the producers frame it on. The following properties capture this intuition. First, we define the CPR of a core predicate $\gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle$ as the memory resulting from its production in $\mathbf{0}$, which must succeed in any satisfiable context:

$$\pi \ \mathtt{SAT} \implies \mathbf{0}.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle, \mathsf{true})^{\mathcal{S}}_\pi \wedge \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle \neq \mathbf{0}.$$

overloading notation for the core predicate and its resource. Moreover, we require that any successful production frames on the CPR:

$$\mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu', \mathsf{true})^{\mathcal{S}}_{\pi'} \implies \mu' = \mu \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle$$

[6] A PCM, $X = \langle X, \bullet, \mathbf{0} \rangle$, comprises a carrier set $X$ (overloaded for simplicity), a partial, associative, and commutative composition operator $\bullet$, and unit element $\mathbf{0}$.

[7] Note that this requirement makes concrete memory actions deterministic.

and also that producers cannot return missing information errors, as they are meant to succeed precisely when the CPR is missing. The consumers, on the other hand, must succeed if and only if the CPR is present in memory:

$$\begin{aligned} \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \pi' \vdash \mu = \mu' \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle \\ \pi \vdash \mu = \mu' \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle \wedge \mathcal{W}f_\pi(\mu) &\implies \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi} \end{aligned}$$

with the resulting context $\pi'$ having enough information to isolate the CPR[8]. Interestingly, erroneous executions cannot be fully characterised in terms of CPR presence or absence, because of TL-specific error cases: for example, in C, attempting to either get or set the value of a block cell that is beyond the block bound raises an out-of-bounds error (cf. §3). What we require instead is that consumed CPR can always be re-produced, that producers fail in a memory in which consumers succeed, and that producers succeed in a memory in which consumers return a missing information error (and vice versa for the latter):

$$\begin{aligned} \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}'_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'} \\ \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} &\implies \mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot -)_\pi \rightsquigarrow (\mu, \mathsf{false})^{\mathcal{E}}_{\pi'} \\ \mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{\mathcal{M}}_{\pi'} &\iff \mu.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu \bullet \gamma\langle \mathsf{v}_i \cdot \mathsf{v}_o \rangle, \mathsf{true})^{\mathcal{S}}_{\pi'} \end{aligned}$$

The properties given so far allow us, for example, to prove that well-formed memories cannot contain duplicated CPR. The final property below requires that non-missing executions of consumers and erroneous executions of producers must be frame-preserving, with the former formulated as follows:

$$\begin{aligned} &\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{r}_{\pi'} \wedge r \neq \mathcal{M} \wedge (\pi'' \Rightarrow \pi') \wedge \mathcal{W}f_{\pi''}(\mu \bullet \mu_f) \\ &\quad \implies (\mu \bullet \mu_f).\mathrm{cons}_\gamma(\mathsf{v}_i)_{\pi''} \rightsquigarrow (\mu' \bullet \mu_f, \mathsf{v}_o)^{r}_{\pi''} \end{aligned}$$

where $\pi''$ effectively maintains well-formedness constraints for $\mu$, adds on further ones required for $\mu \bullet \mu_f$ to be defined and also isolates the consumed CPR. Note that neither missing executions of consumers nor successful executions of producers can be frame preserving, as framing on the appropriate CPR could result in success for the former, and a duplicated resource error for the latter.

Using the consumers and producers, we are able to derive <u>getter</u> and <u>setter</u> actions, $A \triangleq \{\mathrm{get}_\gamma, \mathrm{set}_\gamma : \gamma \in \Gamma\}$, which perform frame-preserving CPR lookup and mutation, as given below. We discuss getters and setters further in §3, in the context of our JS and C instantiations.

Getter: Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', \mathsf{v}_o)^{\mathcal{S}}_{\pi'} \qquad \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}{\mu.\mathrm{get}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu'', \mathsf{v}_o)^{\mathcal{S}}_{\pi'}}$$

Setter: Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu', -)^{\mathcal{S}}_{\pi'} \qquad \mu'.\mathrm{prod}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}{\mu.\mathrm{set}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi'}}$$

Getter: Non-Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'} \qquad r \neq \mathcal{S}}{\mu.\mathrm{get}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'}}$$

Setter: Non-Success

$$\frac{\mu.\mathrm{cons}_\gamma(\mathsf{v}_i)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'} \qquad r \neq \mathcal{S}}{\mu.\mathrm{set}_\gamma(\mathsf{v}_i \cdot \mathsf{v}_o)_\pi \rightsquigarrow (\mu, \mathsf{false})^{r}_{\pi'}}$$

[8] The $\pi \vdash \ldots$ denotes reasoning under context $\pi$. In the concrete case, it can be ignored.

**Compositional State Models.** Compositional memory models lift to compositional state models, in a similar way to the lifting of the complete memory models illustrated in [19]; see [33] for details. Here, we focus on memory action execution, which is lifted as follows to state action execution, given a memory model $M(\mathsf{V}, \Gamma)$ and $\alpha \in A_\Gamma \uplus A$:

$$ea(\alpha, \langle \mu, \rho, \pi \rangle, \mathsf{v}) \triangleq \{(\langle \mu', \rho, \pi' \rangle, \mathsf{v}')^r \mid \mu.\alpha(\mathsf{v})_\pi \rightsquigarrow (\mu', \mathsf{v}')^r_{\pi'}\}.$$

Observe how the context of the state is passed to the memory execution function, which may then strengthen it before passing it back to the resulting state. We can show that the PCM and well-formedness relation on memories lift to a PCM and well-formedness relation on states, and that state action execution maintains properties analogous to those given for memory models.

## 2.2 GIL Verification

We give an overview of Gillian verification based on separation logic (SL); see [33] for details. We describe GIL assertions, parameterised by the core predicates of the TL, define assertion satisfiability in a novel, parametric way using the core predicate producers, and provide a mechanism for using verified function specifications in GIL execution.

**GIL Assertion Syntax**

$$p, q \in \mathcal{A} \triangleq \mathsf{emp} \mid p * q \mid \gamma\langle \hat{e}_i \cdot \hat{e}_o \rangle \mid \delta\langle \hat{e}_i \cdot \hat{e}_o \rangle$$
$$P, Q \in \mathcal{A}srt \triangleq \{p \wedge \pi \mid p \in \mathcal{A}, \pi \in \Pi\}$$
$$pred \in \mathcal{P}red \triangleq \text{pred } \delta\langle \hat{x}_i \cdot \hat{x}_o \rangle := P_1; \ldots; P_n;$$

A compositional memory model with core predicates $\Gamma$ induces an SL-assertion language given on the right. GIL memory assertions, $p, q \in \mathcal{A}$, are formed using the empty assertion, the separating conjunction, the core predicates, and user-defined predicates, whose names come from a dedicated set, $\Delta \ni \delta$. The empty assertion and the separating conjunction are standard. Core predicate assertions are lifted from memory core predicates. User-defined predicates, introduced by example in §3 and §4, are used by tool developers to characterise the interface of the TL, and by code developers to describe the data structures in their programs. They have in- and out-parameters like core predicates, and can have multiple definitions, separated by a semi-colon. Assertions, $P, Q \in \mathcal{A}srt$, extend memory assertions with pure first-order assertions, $\pi$, conflated with Boolean symbolic expressions.

**Satisfiability.** To define assertion satisfiability, we lift memory consumers and producers from core predicates to memory assertions, denoted by $\mu.\text{cons}_\theta(p)$ and $\mu.\text{prod}_\theta(p)$, and then to states and arbitrary assertions, denoted by $\sigma.\text{cons}_\theta(P)$ and $\sigma.\text{prod}_\theta(P)$, using substitutions $\theta : \hat{\mathcal{X}} \mapsto \mathsf{V}$ (extended to symbolic expressions inductively, in the standard way) to map core predicate assertions, with parameters given by symbolic expressions, to the core predicates of the memory model, with parameters given by values. We highlight the successful base case of the memory assertion consumers, where the returned context requires the out-parameters of the assertion to match the ones found in memory:

$$\frac{\mu.\text{cons}_\gamma(\theta(\hat{e}_i))_\pi \rightsquigarrow (\mu', \mathsf{v}'_o)^{\mathcal{S}}_{\pi'} \qquad \pi'' = (\pi' \wedge \mathsf{v}'_o = \theta(\hat{e}_o)) \qquad \pi'' \ \texttt{SAT}}{\mu.\text{cons}_\theta(\gamma\langle \hat{e}_i \cdot \hat{e}_o \rangle)_\pi \rightsquigarrow (\mu', \mathsf{true})^{\mathcal{S}}_{\pi''}}$$

and the successful consumption of an arbitrary assertion $P = p \wedge \pi$:

$$\frac{\mu'.\mathrm{cons}_\theta(p)_{\pi'} \rightsquigarrow (\mu'', \mathsf{true})^{\mathcal{S}}_{\pi''} \qquad \pi'' \vdash \theta(\pi)}{\langle \mu', \rho, \pi' \rangle.\mathrm{cons}_\theta(p \wedge \pi) \rightsquigarrow (\langle \mu'', \rho, \pi'' \rangle, \mathsf{true})^{\mathcal{S}}}$$

**Definition 4 (Satisfiability).** *The satisfiability relation, stating that memory $\mu'$ and context $\pi'$ satisfy assertion $p \wedge \pi$ under substitution $\theta$, is defined by:*

$$\mu', \pi', \theta \models p \wedge \pi \iff \mathbf{0}.\mathit{prod}_\theta(p)_{\mathsf{true}} \rightsquigarrow (\mu_p, \mathsf{true})^{\mathcal{S}}_{\pi_p} \wedge \pi' \vdash (\mu' = \mu_p \wedge \pi_p \wedge \theta(\pi))$$

*and is lifted to states as:* $\langle \mu', \rho, \pi' \rangle, \theta \models p \wedge \pi$ *if and only if* $\mu', \pi', \theta \models p \wedge \pi$.

In Definition 4, the production, when successful, creates the (unique) memory $\mu_p$ that corresponds to the resource of the assertion $p$, with its (unique) well-formedness constraints, $\pi_p$. In the concrete case, as the only allowed context is $\mathsf{true}$, the formulation simplifies to the more intuitive $\mathbf{0}.\mathrm{prod}_\theta(p) \to (\mu', \mathsf{true})^{\mathcal{S}} \wedge \theta(\pi)$.

**Specifications.** Gillian function specifications have the form $\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}}$, where $f$ is the function identifier, $x$ is the function parameter, $\hat{x}$ is the symbolic variable holding the value of $x$, $P$ is the pre-condition, $Q$ is the post-condition, and $\hat{e}$ is the return value of the function, with the following, well-known, constraints:

1. program variables do not appear in the pre- or the post-condition, and the function parameter $x$ is accessed using the symbolic variable $\hat{x}$;
2. symbolic variables that appear in a pre-condition are implicitly universally quantified, and can be re-used in the corresponding post-condition; and
3. symbolic variables that appear only in a post-condition are implicitly existentially quantified.

We extend GIL programs with function specifications, accessible via $\mathsf{p}.\mathrm{specs}$, and the GIL execution semantics with rules for folding and unfolding user-defined predicates, as well as with a rule for calling function specifications, the success case of which is given below. Gillian verifies a specification $\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}}$ if, given the identity substitution $\hat{\theta}$ and a symbolic state $\hat{\sigma}$ with store $\{x \mapsto \hat{\theta}(\hat{x})\}$ such that $\hat{\sigma}, \hat{\theta} \models P$, the symbolic execution of $f$ starting from $\hat{\sigma}$ always terminates, for all final symbolic states $\hat{\sigma}_i$ there exists some $\hat{\theta}_i \geq \hat{\theta}$ such that $\hat{\sigma}_i, \hat{\theta}_i \models Q$, and the corresponding return value equals $\hat{\theta}_i(\hat{e})$ under the context of $\hat{\sigma}_i$. We can prove that if Gillian verifies a specification, then its standard SL interpretation holds.

Spec Call - Success

$$\frac{\begin{array}{lr}
\mathsf{cmd}(\mathsf{p}, cs, i) = y := e(e') \text{ with } \theta & \text{function call with substitution } \theta \\
\sigma.\mathrm{eval}_e(-) \rightsquigarrow f \quad \sigma.\mathrm{eval}_{e'}(-) \rightsquigarrow \mathsf{v}' & \text{get function id and parameter value} \\
\{\hat{x}, P\} f(x) \{Q\}^{\hat{e}} \in \mathsf{p}.\mathrm{specs} & \text{get one of the function specifications} \\
\theta' = \theta[\hat{x} \mapsto \mathsf{v}'] & \text{extend substitution with parameter value} \\
\sigma.\mathrm{cons}_{\theta'}(P) \to \{(\sigma_j, \mathsf{true})^{\mathcal{S}}|_{j \in J}\} & \text{consume pre-condition} \\
j \in J & \text{select a branch} \\
\sigma_j.\mathrm{prod}_{\theta'}(Q) \rightsquigarrow (\sigma'_j, \mathsf{true})^{\mathcal{S}} & \text{produce post-condition} \\
\sigma'_j.\mathrm{setVar}_y(\theta'(\hat{e})) \rightsquigarrow \sigma' & \text{assign return value}
\end{array}}{\mathsf{p} \vdash \langle \sigma, cs, i \rangle \rightsquigarrow \langle \sigma', cs, i{+}1 \rangle}$$

Note that for this rule to succeed, the consumption of $P$ must succeed. The rule is slightly simplified for presentation. First, it assumes to have the substitution upfront; in the implementation, we have a unification algorithm that, starting from the function parameter and using the consumers, learns the substitution. Second, it assumes that the post-condition does not introduce fresh symbolic variables; these are handled using allocators and added to the substitution.

**Remark.** Due to space constraints, we have not been able to give the full technical details of Gillian verification. These are available in the Gillian technical report [33], where we demonstrate that the overall GIL execution using compositional memory models is frame-preserving (up to the usual renaming of allocated memory locations) and prove a standard verification soundness result.

## 3 Compositional Memory Models: JavaScript and C

We present the compositional memory models of JS and C, giving the basic actions and core predicates, and some of the user-defined predicates that capture the intuitive interfaces of these languages. The key ideas behind compositional JS memory models were introduced in the JaVerT project [21,20,22]; we transfer them to Gillian. We introduce the compositional C memory models, building on the concrete block-offset memory model of CompCert [31], simplifying the presentation.[9] In doing so, we highlight a striking similarity between the JS and C models that is the result of our emphasis on negative resource.

The JS and C concrete compositional memory models are made up of building blocks that are assigned a unique location (or identifier) from a set of uninterpreted symbols, $\mathcal{L} \subset \mathcal{U}$: for JS, the building blocks are the extensible objects; for C, they are the blocks of linear memory of a given size. Each building block is divided into at least one component. For JS, each object has three components: a property table, $h : \mathcal{S} \rightharpoonup \mathcal{V}al$, partially mapping property names (strings) to values; a domain, $d : \mathcal{P}(\mathcal{S})$, discussed shortly; and metadata, $m : \mathcal{V}al$, which keeps track of internal JS properties for that object [22]. For C, each block has two components: the block contents $k : \mathbb{N} \rightharpoonup \mathcal{V}al$, partially mapping offsets (natural numbers) to values; and a bound, $n : \mathbb{N}$, discussed shortly. Finally, the memory units are, intuitively, the parts of the memory components that cannot be separated further: for JS, these are single object properties, domains, and metadata; for C, these are single block cells and bounds. These memory units directly correspond to the core predicates given in Definitions 6 and 7.

Compositional memory models must keep track of negative resource, which can come from two sources: allocation and deallocation. For JS and C, the negative information originating from allocation has infinite representation: in JS, a freshly created object is known to not have any properties; in C, a freshly allocated block of a given size in C is known not to have offsets beyond that size. This infinite information is captured, for JS, by the object domain whose meaning

[9] We assume that values have the same size in memory and omit permissions. Gillian-C implements the full models, eliding the concurrency-related aspects of permissions.

is that any property not in the domain is absent, and, for C, by the block bound whose meaning is that any accesses beyond that bound result in a buffer overrun error. The negative information originating from deallocation is easier to handle, tracked by a dedicated uninterpreted symbol, $\varnothing \in \mathcal{U}$. In JS, deallocation is at the unit level: only object properties are deleted. This is captured by extending the co-domain of property tables with $\varnothing$: that is, $h : \mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}$. In C, deallocation is at the building-block level: only entire blocks can be deleted. This is captured by extending the co-domain of blocks with $\varnothing$, indicating that a block has been freed.

Due to compositionality, any building block, component or unit can be missing. In the theory, we capture this either implicitly, via absence from the domain of a mapping (e.g., a missing object property for JS or a missing block cell for C), or explicitly, using the symbol $\bot$ (e.g. a missing domain, metadata, or bound).

**Definition 5 (Compositional JS and C Memories).** *The PCMs of compositional concrete JS and C memories, $|M_{JS}|$ and $|M_C|$, are given by the sets*

$$\begin{aligned} \mu \in |M_{JS}| &: \mathcal{L} \rightharpoonup ((\mathcal{S} \rightharpoonup \mathcal{V}al_{\varnothing}) \times \mathcal{P}(\mathcal{S})_{\bot} \times \mathcal{V}al_{\bot}), \\ \mu \in |M_C| &: \mathcal{L} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}al) \times \mathbb{N}_{\bot})_{\varnothing}, \end{aligned}$$

*composition defined as disjoint union, and empty memory $\emptyset$. The PCMs of compositional symbolic JS and C memories, $|\hat{M}_{JS}|$ and $|\hat{M}_C|$, are given by the sets*

$$\begin{aligned} \hat{\mu} \in |\hat{M}_{JS}| &: \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\bot} \times \hat{\mathcal{E}}xpr_{\bot}), \\ \hat{\mu} \in |\hat{M}_C| &: \hat{\mathcal{E}}xpr \rightharpoonup ((\hat{\mathcal{E}}xpr \rightharpoonup \hat{\mathcal{E}}xpr) \times \hat{\mathcal{E}}xpr_{\bot})_{\varnothing}, \end{aligned}$$

*with composition defined as (syntactic) disjoint union, and empty memory $\emptyset$.*

In the above definition, symbolic memory models are simple liftings of the concrete ones. In the implementation, we employ heavy optimisation: for example, in Gillian-C, we have developed a complex tree representation of symbolic blocks inspired by [29], enabling tractable reasoning about arrays of symbolic size.

Well-formedness of concrete memories addresses the relationship between positive and negative information, given for JS and C below:

$$\begin{aligned} \mathcal{W}f^{\text{JS}}(\mu) &\triangleq \forall (h, d, -) \in \mathsf{codom}(\mu).\ d \neq \bot \implies \mathsf{dom}(h) \subseteq d \\ \mathcal{W}f^{\text{C}}(\mu) &\triangleq \forall (k, n) \in \mathsf{codom}(\mu).\ n \neq \bot \implies \mathsf{dom}(k) \subseteq [0, n) \end{aligned}$$

Well-formedness of symbolic memories additionally has to address separation of locations and separation in any other mappings with symbolic expressions in its domain (e.g. object properties for JS and offsets for C). We give the well-formedness criterion for the symbolic C memory:

$$\hat{\mathcal{W}f}^{\text{C}}_{\pi}(\hat{\mu}) \triangleq \pi \vdash \bigwedge_{\substack{\hat{l}, \hat{l}' \in \mathsf{dom}(\hat{\mu}) \\ \hat{l} \not\equiv \hat{l}'}} \hat{l} \neq \hat{l}' \wedge \bigwedge_{\substack{(\hat{k}, -) \in \mathsf{codom}(\hat{\mu}) \\ \hat{o}, \hat{o}' \in \mathsf{dom}(\hat{k}), \hat{o} \not\equiv \hat{o}'}} \hat{o} \neq \hat{o}' \wedge \bigwedge_{\substack{(\hat{k}, \hat{n}) \in \mathsf{codom}(\hat{\mu}) \\ \hat{o} \in \mathsf{dom}(\hat{k}), \hat{n} \not\equiv \bot}} \hat{o} < \hat{n}$$

For our JS and C instantiations, the core predicates follow straightforwardly from the units of their memory models.

$$\textsc{CConsCell - Found}\quad \frac{\mu(l) = (k, n) \qquad k(o) = v \\ k' = k \setminus \{o\} \qquad \mu' = \mu[l \mapsto (k', n)]}{\mu.\mathsf{consCell}([l, o]) \rightsquigarrow (\mu', v)^{\mathcal{S}}}$$

$$\textsc{SConsCell - Use After Free}\quad \frac{\hat{\mu}(\hat{l}') = \varnothing \qquad \pi' = (\hat{l} = \hat{l}') \qquad (\pi \wedge \pi')\ \texttt{SAT}}{\hat{\mu}.\mathsf{consCell}\ ([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}, \mathsf{false})^{\mathcal{E}}_{\pi \wedge \pi'}}$$

$$\textsc{SConsCell - Found}\quad \frac{\hat{\mu}(\hat{l}') = (\hat{k}, \hat{n}) \quad \hat{k}(\hat{o}') = \hat{v} \\ \pi' = ([\hat{l}, \hat{o}] = [\hat{l}', \hat{o}']) \quad (\pi \wedge \pi')\ \texttt{SAT} \\ \hat{k}' = \hat{k} \setminus \{\hat{o}'\} \quad \hat{\mu}' = \hat{\mu}[\hat{l}' \mapsto (\hat{k}', \hat{n})]}{\hat{\mu}.\mathsf{consCell}\ ([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}', \hat{v})^{\mathcal{S}}_{\pi \wedge \pi'}}$$

$$\textsc{SConsCell - Missing Cell}\quad \frac{\hat{\mu}(\hat{l}') = (\hat{k}, \hat{n}) \\ \pi_k = (\hat{l} = \hat{l}') \wedge \hat{o} \notin \mathsf{dom}(\hat{k}) \\ \pi_n = (\hat{n} = \bot) \vee (\hat{n} \geq \hat{o}) \quad (\pi \wedge \pi_k \wedge \pi_n)\ \texttt{SAT}}{\hat{\mu}.\mathsf{consCell}\ ([\hat{l}, \hat{o}])_{\pi} \rightsquigarrow (\hat{\mu}, \mathsf{false})^{\mathcal{M}}_{\pi \wedge \pi_k \wedge \pi_n}}$$

Fig. 2: Selected rules for the consCell consumer.

**Definition 6 (JS Core Predicates).** *JS has three core predicates, $\gamma_{JS} \in \Gamma_{JS}$:*
- *the object-property predicate, $(\hat{l}, \hat{p}) \mapsto \hat{v}$, which states that property $\hat{p}$ of object at location $\hat{l}$ contains value $\hat{v}$ (including $\varnothing$ denoting property absence);*
- *the domain predicate,* $\mathsf{domain}(\hat{l}, \hat{d})$, *which states that object at location $\hat{l}$ has no properties outside the finite set $\hat{d}$;*
- *the metadata predicate,* $\mathsf{metadata}(\hat{l}, \hat{m})$, *which states that object at location $\hat{l}$ has metadata $\hat{m}$.*

**Definition 7 (C Core Predicates).** *C has three core predicates, $\gamma_C \in \Gamma_C$ [10]:*
- *the cell predicate, $(\hat{l}, \hat{o}) \mapsto \hat{v}$, which states that the cell at offset $\hat{o}$ in the block at location $\hat{l}$ contains value $\hat{v}$ (which, this time, does not include $\varnothing$);*
- *the bounds predicate,* $\mathsf{bound}(\hat{l}, \hat{n})$ *, which states that any cell beyond offset $\hat{n}$ in block at location $\hat{l}$ is not there;*
- *the freed predicate, $\hat{l} \mapsto \varnothing$, which states that block at location $\hat{l}$ has been freed.*

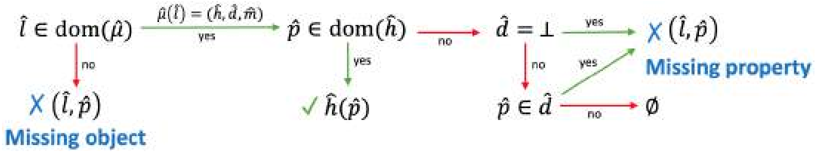
We illustrate the C predicate action execution functions, $\underline{\mathsf{ea}}_{\,C}$ and $\underline{\hat{\mathsf{ea}}}_{\,C}$, respectively, with a selection of rules for the C cell-predicate consumer, consCell, given in Figure 2. The remaining rules, as well as the rules for their JS counterparts, $\underline{\mathsf{ea}}_{\,JS}$ and $\underline{\hat{\mathsf{ea}}}_{\,JS}$, can be found in the Gillian technical report [33]. With this information, we can define the compositional concrete and symbolic JS and C memory models.

**Definition 8 (JS Memory Models).** *The compositional concrete and symbolic JS memory models are defined, respectively, as $M_{JS}(\mathcal{V}al, \Gamma_{JS}) = \langle |M_{JS}|, \mathcal{W}f^{JS}, \underline{\mathsf{ea}}_{\,JS}\rangle$ and $\hat{M}_{JS}(\hat{\mathcal{E}}xpr, \Gamma_{JS}) = \langle |\hat{M}_{JS}|, \hat{\mathcal{W}f}^{JS}, \underline{\hat{\mathsf{ea}}}_{\,JS}\rangle$.*

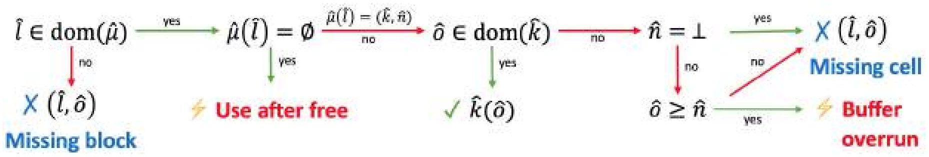
**Definition 9 (C Memory Models).** *The compositional concrete and symbolic C memory models are defined, respectively, as $M_C(\mathcal{V}al, \Gamma_{JS}) = \langle |M_C|, \mathcal{W}f^{C}, \underline{\mathsf{ea}}_{\,C}\rangle$ and $\hat{M}_C(\hat{\mathcal{E}}xpr, \Gamma_{JS}) = \langle |\hat{M}_C|, \hat{\mathcal{W}f}^{C}, \underline{\hat{\mathsf{ea}}}_{\,C}\rangle$.*

[10] In full C and the Gillian-C implementation, memory values may be of different sizes, and holes may exist between these values due to alignment restrictions. To address this, the implemented cell assertion carries additional information related to, e.g., size and type, similarly to that of [4], and there also exists a `hole` core predicate.

The getters and setters for JS and C are defined using the methodology described in §2. In particular, the JS getters and setters are given by $A_{\text{JS}} = \{\mathsf{getProp}, \mathsf{setProp}, \mathsf{getDomain}, \mathsf{setDomain}, \mathsf{getMetadata}, \mathsf{setMetadata}\}$, and the summary of the execution of the symbolic $\mathsf{getProp}(\hat{l}, \hat{p})$ getter is illustrated below:



Similarly, the C getters and setters are given by $A_{\text{C}} = \{\mathsf{getCell}, \mathsf{setCell}, \mathsf{getBound}, \mathsf{setBound}, \mathsf{getFreed}, \mathsf{setFreed}\}$ and the summary of the execution of the symbolic $\mathsf{getCell}(\hat{l}, \hat{o})$ getter is illustrated below:



The similarities in the two diagrams are evident, with the main difference being that JS getters do not throw errors, whereas C getters do.

**User-defined JS and C Predicates.** Core predicates describe fundamental units of the TL memory model. On top, user-defined predicates build layers of abstraction to describe memory components and building blocks, standard library interfaces, all the way to complex data structures for particular code such as the AWS message header. Using Gillian notation, we present some of the JS and C user-defined predicates; in this notation: $*$ and $\wedge$ are conflated to $*$, with automatic differentiation between spatial and pure assertions[11]; predicate definitions are separated with a semi-colon; and logical variables are prefixed with the # symbol and are implicitly existentially quantified in predicate definitions.

Gillian-JS inherits many user-defined predicates from JaVerT [21], including simple ones for describing JS objects and their properties, as well as advanced ones for specifying scoping, function closures and prototype chains. We focus here on the new `FrozenObject(o, proto, pvs)` predicate, which describes a frozen object[12] `o` with prototype `proto` and property-value pairs `pvs`. We first define the predicate `FrozenObjectProps(o, pvs)` to grab the resource of the object properties:

```
pred FrozenObjectProps(o, pvs) : pvs = [ ];
    pvs = [#p, #v] :: #rpvs * DataPropConst(o, #p, #v) *
    FrozenObjectProps(o, #rpvs);
```

where `DataPropConst(o, #p, #v)` states that the object `o` has a non-writable property `#p` with value `#v`. We then add information about the object prototype and its non-extensibility using the `JSObject(o, proto, ext)` predicate, and also state that the object has no properties other than `pvs` using the domain core predicate:

[11] From the separation logic literature, the pure assertions can be regarded as dotted.
[12] A JS object is frozen if it cannot be extended and all its properties are non-writable.

```
pred FrozenObject(o, proto, pvs) :
    JSObject(o, proto, false) * FrozenObjectProps(o, pvs) *
    FirstProj(pvs, #ps) * ListToSet(#ps, #pss) * domain(o, #pss)
```

where `FirstProj`(`pvs`, #`ps`) means that the list #`ps` is the first projection of the list of pairs `pvs`, and `ListToSet`(#`ps`, #`pss`) means that the elements of the list #`ps` form the set #`pss`.

Gillian-C, on the other hand, comes with user-defined predicates capturing, for example, arrays and blocks in memory, as well as automatically-generated predicates describing C structs, with support for nested structs. In particular, the `array`(`b`, `off`, `c`) predicate describes a contiguous fragment of a block `b`, starting from offset `off`, with contents described by the mathematical list `c`:

```
array(b, off, c) : c = [];
                   (b, off) -> #c * array(b, off+1, #d) * c = #c :: #d
```

and the `block`(`b`, `c`) predicate captures an entire C block with contents `c`:

```
block(b, c) : array(b, 0, c) * bound(b, |c|)
```

In the implementation, arrays also exist as core predicates. This allows us to reason about arrays automatically in the symbolic memory (e.g., to split an array into sub-arrays), supported by our tree representation of symbolic blocks, instead of requiring manual application of lemmas.

Finally, we illustrate automatically generated struct-related predicates using the `aws_byte_cursor` structure given below, which contains two fields: an unsigned integer `len`; and a nullable pointer to an array of 8-bit unsigned integers `buf`. This struct is used for traversing the AWS message header (cf. §4), and is intended to capture an array in memory that starts at `buf` and has length `len`.

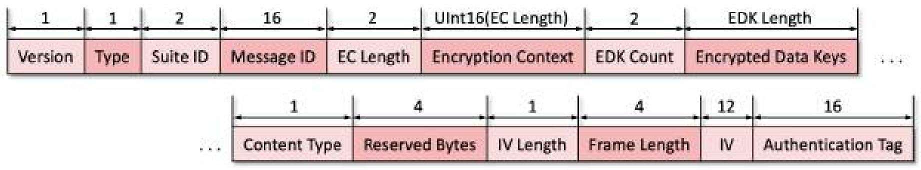
```
struct aws_byte_cursor {  pred struct_aws_byte_cursor(cur, len, buf) :
  size_t len;               (cur == [#b, #o]) * ((#b, #o) -int64-> len) *
  uint8_t *buf;             ((#b, #o +p 8) -int64-> buf) *
}                           is_ptr_or_null(buf)
```

The generated predicate describes the struct's layout in memory and gives basic typing information: it states that an `aws_byte_cursor`, starting from the position given by the pointer `cur`, occupies 16 bytes in memory (8 + 8, given by the type annotation `int64`), with the first 8 bytes taken by `len`, and the second 8 bytes (note the pointer addition +`p`) taken by `buf`, which is either a pointer or `null`.

## 4 AWS Encryption SDK Message Header Specification

The encrypted data handled by the AWS Encryption SDK is stored within a structure called a message [3]. The message format has two versions of similar complexity: we verify version 1; version 2 was introduced very recently. Messages consist of a header, a body, and a footer. Here, we describe only the structure of the header, as we are verifying header deserialisation.

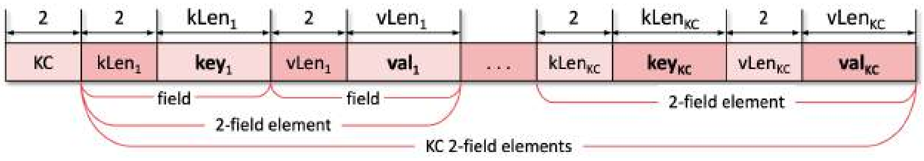
The AWS Encryption SDK message header is a sequence of bytes (buffer) divided into sections, as illustrated below; above each section is its length in bytes.



Our approach is to abstract the header contents into a list and formulate pure predicates that describe its structure in a language-independent way. This allows us to then use the same abstractions as part of further, language-dependent, abstractions for both JS and C. Our design of the abstractions was informed by existing code annotations found in the implementations, which describe simple first-order properties of the code and, in the case of C, can also link to the CBMC [30] bounded model checker. However, these annotations are limited by the expressivity of JS and C, particularly when it comes to reflecting on the memory contents. Our predicates have no such limitations.

We narrow down our exposition to the encryption context, as it illustrates well the language-independent and language-dependent aspects of our specification, and is also the section in which we discovered bugs in both implementations.

**Pure Specification of the Encryption Context.** The encryption context (EC) is a sequence of bytes that describes a set of key-value pairs. Its structure is given in the diagram below.

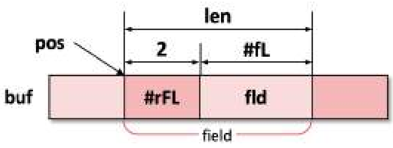


The first two bytes represent the number of key-value pairs, denoted by `KC`, and the rest describe the `KC` key-value pairs themselves. Keys and values are represented by sequences of bytes and, as they are of variable length, are serialised by first having *two bytes* that represent the length, followed by *that many bytes* of the actual key or value; we refer to this pattern as a field, and to a sequence of $n$ fields as an $n$-element. Then, a key-value pair is serialised as a 2-field element, and all of the key-value pairs form a sequence of `KC` 2-field elements.

We specify the EC by building layers of abstraction, from fields to elements to element sequences to the EC, each of which can either be complete, incomplete (partial, but with correct structure), or malformed (with incorrect structure). In the implementation, these are specified separately and are joined together in appropriate over-arching abstractions. Here, we focus on complete variants only.

The `Field`(`buf`, `pos`, `fld`, `len`) predicate states that the buffer (list of bytes) `buf`, at index `pos`, holds a field with contents `fld` (list of bytes) and total length `len`:

```
pred Field(buf, pos, fld, len) :
 (0 <= pos) * (#rFL = sub(buf, pos, 2)) *
 UInt16(#rFL, #fL) *
 (fld = sub(buf, pos+2, #fL)) *
 (len = 2+#fL) * (pos+len <= |buf|)
```



This predicate uses the GIL operator `sub(l, s, n)`, which returns the sublist of list `l` starting from index `s` and of length `n`, and also the `UInt16(rn, n)` predicate, which states that `n` is a 16-bit big-endian interpretation of the raw 2-byte list `rn`. The `Element(buf, pos, fC, elem, len)` predicate states that buffer `buf` at index `pos` holds a sequence of `fC` fields, with contents `elem` (a list of the appropriate field contents) and total length `len`. It is defined similarly to a standard linked-list predicate, with the 'link' being the fact that the list members are contiguous in memory:

```
pred Element(buf, pos, fC, elem, len) :
  (fC = 0) * (0 <= pos) * (pos <= |buf|) * (elem = [ ]) * (len = 0);
  (0 < fC) * Field(buf, pos, #fld, #fL) * Element(buf, pos+#fL, fC-1, #rFs,
  #rL) * (elem = #fld :: #rFs) * (len = #fL+#rL)
```

Next, analogously to `Element`, we define the `Elements(buf, pos, eC, fC, elems, len)` predicate, which states that the buffer `buf`, at index `pos`, holds a sequence of `eC` elements, each with `fC` fields, with contents `elems` (a list of the appropriate element contents) and of total length `len`. Finally, the `EncryptionContext(buf, KVs)` predicate states that the entire buffer `buf` is an EC with key-value pairs `KVs`, with all keys being unique:

```
pred EncryptionContext(buf, KVs) : (buf = [ ]) * (KVs = [ ]);
     (#rKC = sub(buf, 0, 2)) * UInt16(#rKC, #KC) * (0 < #KC) *
     Elements(buf, 2, #KC, 2, KVs, #len) *
     FirstProj(KVs, #Ks) * Unique(#Ks) * (2+#len = |buf|)
```

Next, we show how this pure specification of the EC contents can be connected without modification to both the JS and C memories.

**Encryption Context in JS.** In JS, the EC is serialised as an ArrayBuffer, which is a raw binary data buffer in memory, and accessed using a Uint8Array, which is a view on top of that ArrayBuffer starting from a given offset and of a given length, treating the raw data underneath as 8-bit unsigned integers. This Uint8Array view is similar in function to the `aws_byte_cursor` C structure (cf. §3). Abstracting ArrayBuffer contents to lists, we connect these data structures in JS memory to our pure EC specification (cf. Figure 3, top and centre):

```
pred JSSerEC(o, EC, KVs) :
     Uint8Array(o, #aBuf, #off, #len) * ArrayBuffer(#aBuf, #data) *
     (EC = sub(#data, #off, #len)) * EncryptionContext(EC, KVs)
```

In JS, the EC is deserialised into a frozen JS object with prototype `null`, whose properties represent the keys and hold the values. This is done by converting the keys and the values to UTF-8 strings, and is specified as follows:

```
pred JSDeserEC(o, KVs) : toUtf8(KVs, #sKVs) * FrozenObject(o, null, #sKVs)
```

where `toUtf8` converts the list `KVs` point-wise to strings, obtaining #`sKVs`.

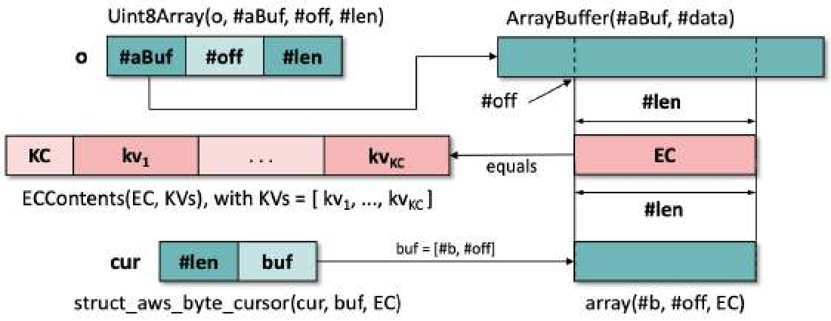

Uint8Array(o, #aBuf, #off, #len)
o
#aBuf
#off
#len
ArrayBuffer(#aBuf, #data)
#off
#len
KC
kv₁
...
kv_KC
equals
EC
ECContents(EC, KVs), with KVs = [ kv₁, ..., kv_KC ]
#len
cur
#len
buf
buf = [#b, #off]
struct_aws_byte_cursor(cur, buf, EC)
array(#b, #off, EC)

Fig. 3: Serialised Encryption Context: language-independent pure part (red; middle) and language-specific resource (green; JS above, C below)

Finally, the specification of the `decodeEncryptionContext` function states that the EC deserialisation is performed correctly.

```
{ JSSerEC(eEC, #EC, #KVs)                       }
  function decodeEncryptionContext(eEC)
{ PRE-CONDITION * JSDeserEC(ret, #KVs)  }
```

**Encryption Context in C.** In C, the EC is serialised as a block in memory, and is traversed using an AWS byte cursor. Using the auto-generated predicate given in §3, we define the `aws_byte_cursor`(`cur`, `buf`, `c`) predicate, stating that `cur` points to a byte cursor which has access to an array starting from `buf`, and holding contents `c`, making the length implicit:

```
  pred aws_byte_cursor(cur, buf, c) :
    struct_aws_byte_cursor(cur, #len, buf) * (buf = [#b, #off]) *
    array(#b, #off, c) * (#len = |c|)
```

A serialised EC can then be described as a valid byte cursor whose contents represent the EC key-value pairs (cf. Figure 3, centre and bottom):

```
pred CSerEC(cur, buf, EC, KVs) :
  aws_byte_cursor(cur, buf, EC) * EncryptionContext(EC, KVs)
```

In C, the EC is deserialised into an AWS hash table, whose keys and values directly correspond to the key/value pairs of the EC, specified as follows, eliding the internal structure of the hash tables due to space constraints:

```
pred CDeserEC(ht, KVs) : valid_hash_table(ht, KVs)
```

The specification of the EC deserialisation function is more complex than for JS. In particular, the byte cursor that originally pointed to the EC ends up shifted to the end of the byte buffer, exposing the array underneath the `CSerEC` predicate.

```
{  empty_hash_table(ec) * CSerEC(cur, #buf, #EC, #KVs)            }
   int aws_cryptosdk_enc_ctx_deserialize(
       struct aws_hash_table *ec, struct aws_byte_cursor *cur)
{  (ret = 0) * CDeserEC(ec, #KVs) * (#buf = [#b, #off]) *
   array(#b, #off, #EC) * aws_byte_cursor(cur, #buf +p |#EC|, [ ])  }
```

## 5 AWS Encryption SDK Message Header Verification

Using Gillian-JS and Gillian-C, together with the specifications given in §4, we verify full functional correctness of the header deserialisation module of the AWS Encryption SDK JS [2] (~200loc) and C [1] (~950loc) implementations. In particular, we verify that the deserialisation of a complete header is correct, and the deserialisation of an incomplete or a malformed header raises an appropriate error.

**Verification Effort and Performance.** The JS verification took 3 person-months and the C verification took 2 person-months, with the latter taking less time because a large part of the infrastructure developed for JS could be re-used. We substantially improved the first-order solver of Gillian to reason automatically about complex operations on lists of symbolic length, first used in the modelling of JS ArrayBuffers and then for C dynamic arrays. We created a collection of language-independent predicates and lemmas about their inductive properties (~1.2kloc) that cover the project-specific AWS header, but also re-usable first-order concepts such as list element uniqueness, projections of lists of pairs, conversion from bytes to numbers, and conversion from raw bytes to strings. Similarly, we also had to create language-dependent abstractions and associated lemmas for the JS and C manipulation of the AWS message header (~1.2kloc). Finally, we had to: annotate the code with specifications and loop invariants, with the latter often having more than twenty components; manually apply lemmas to prove numerous complex entailments; and manually unfold user-defined predicates at times (the folding is automated) (~1.1kloc).

On a machine with an Intel Core i7-4980HQ CPU 2.80 GHz, DDR3 RAM 16GB, and a 256GB solid-state hard-drive running macOS, the JS verification takes approximately 45 seconds and the C verification takes approximately six minutes. The C time is longer, in part due to the larger codebase, but mainly due to the complexity of the implementation of the full C memory model, which is able to reason about arrays of symbolic size. This requires frequent satisfiability checks and (for the moment) branching on non-zero array size. These times could both be improved with the implementation of basic merging techniques.

**JS Verification: Bugs/Improvements.** We discovered two bugs and improved one function implementation to link better with the underlying data structure.

- In the `decodeEncryptionContext` function, the object representing the deserialised EC originally had prototype `Object.prototype` which, in this case, due to the prototype inheritance of JavaScript, meant that if an EC key coincided with a property of `Object.prototype`, an error would be thrown incorrectly. This bug was predicted theoretically in [21], and has since been found in several real-world libraries [42], including `cash` and `jQuery`.
- In the same function, in one of the branches the deserialised EC was returned non-frozen, which constituted a potential vulnerability in that third parties could alter non-secret, but authenticated data.
- The `readElements(eC, fC, buf, pos)` function, which reads `eC` elements with `fC` fields from buffer `buf` at index `pos` into a JS array of arrays, was misaligned

with the underlying data structures. Its parameters were non-intuitive (it received `eC·fC`, `buf`, and `pos`), and used complex array operations to re-form the final return value. We re-implemented this function to construct the returned array of arrays efficiently, simplifying specification and verification, and our implementation was integrated into the codebase.

**JS Verification: Caveats.** Our JS verification is correct up to the following caveats. First, as the AWS SDK JS implementation is written in TypeScript, we elide types to obtain JS; this could be automated, potentially generating predicates from the types. Next, some ES6 features, such as patterns in function parameters, are not yet supported by Gillian-JS; these we rewrite to ES5 Strict, preserving their meaning. Next, we use axiomatic specifications of the ArrayBuffer, DataView, and UInt8Array ES6 built-in libraries, as well as of the `Object.freeze` and `Array.prototype.map` built-in functions. These would ideally be accompanied with implementations, tested against the official Test262 test suite [16] and verified against their specifications. Finally, as Gillian does not support higher-order reasoning, we axiomatise the `toUtf8` function, passed into the deserialisation module as a parameter, as an injective function from raw bytes to JS strings.

**C Verification: Bugs.** We discovered three bugs: one logical error; one undefined behaviour; and one over-allocation.

– The deserialisation of the EC mishandled the case when there is not enough data to read it entirely, continuing to read the EDK instead of reporting an error. This allows some malformed headers to be parsed as well-formed.
– The function `aws_byte_cursor_advance`, when called with a `NULL` cursor and a length of 0, resulted in `NULL` + `0` being computed, which is undefined behaviour, although not problematic for most compilers.
– The deserialised EC was stored using `aws_string`, which extends C strings with certain metadata. It is implemented using a structure that includes a flexible array member. We discovered that string creation over-allocated this array by 8 bytes, because our (correct) predicate describing `aws_string`s was not allowing the verification to go through.

**C Verification: Caveats.** Our C verification is correct up to the following caveats. First, we do not use the `aws_byte_cursor_advance_nospec` function, which advances the byte cursor, but also uses complex computation to protect against the Spectre bug. We instead use `aws_byte_cursor_advance`, which has equivalent behaviour, as our specifications are not expressive enough to capture this distinction. Next, we axiomatise the functions of the AWS hash tables and array list libraries, as their verification is of comparable complexity to the entire deserialisation module. Finally, the AWS allocators of the C implementation, which are passed into some of the functions, contain pointers to memory management functions; this is higher-order in nature. In the verification, we assume those functions are `malloc`, `calloc`, and `realloc`.

## 6 Related Work

The literature explores many techniques and tools for verifying JS [44,18,22,21] and C [23,26,28,13,7]. We describe: multi-language verification architectures; JS and C verification tools based on separation logic; C memory models related to our models; and other analyses applied to the AWS Encryption SDK.

**Multi-Language Verification Architectures.** The multi-language verification architectures closest to Gillian are coreStar [6] and Viper [36,35]. Both of these architectures were designed to serve as verification back-ends for TLs and both have at their core a simple intermediate representation with a dedicated symbolic execution engine[13]. However, they work with the TL in different ways.

In coreStar, TL core assertions are modelled as abstract predicates and memory actions as function calls. The function specifications play the role of our consumer and producer actions. The user also has to provide logical axioms, describing properties of the abstract predicates. The Gillian equivalent of these axioms are the implementations of the memory actions using consumers and producers, which can be optimised, but require understanding of the inner workings of Gillian. Like Gillian, coreStar's symbolic execution engine is parametric on the underlying logical theory and can thus be used to reason about any memory model representable using abstract predicates. It is, however, unclear how efficiently this can be done. coreStar has been used inside the tool jStar [15], which has verified implementations of several Java design patterns but was not pushed to more complex Java code. In [21], the authors observed that coreStar was not able to handle tractably even simple JS programs.

Unlike Gillian and coreStar, Viper [35,36] comes with a fixed intermediate language, also called Viper. The user encodes their memory model and corresponding core assertions into the memory model and assertion language of Viper. A key advantage of Viper lies in its expressive permission model, which includes fractional, recursive, and abstract read permissions, as well as in its support for custom mathematical domains, which enable users to extend Viper with their own first-order theories, tailored to the data structures at hand. Viper has mechanisms similar to our consumer and producer actions, called *inhale* and *exhale*. Viper can reason about both sequential and concurrent programs, and has been used to verify programs written in Java, Go, Rust, and Python, but not JS and C. In fact, it is not clear to us how difficult it would be to use Viper to reason about JS objects and the linear memory of C, as neither can be simply expressed using the static objects natively provided by Viper.

**Semi-automatic JS and C Verification Tools.** There are very few verification tools for JS based on separation logic. For example, JaVerT [21] has been used to verify simple sequential data-structure algorithms. Its successor, JaVerT 2.0 [22], provides whole-program symbolic testing, verification and bi-abductive reasoning [10], unified by a core symbolic execution engine.

[13] Viper includes both a symbolic execution engine and a verification condition generator based on Boogie [5] for its intermediate language.

JaVerT 2.0 verification is more efficient than JaVerT verification, but has still only been applied to simple data-structure algorithms. Gillian [19] builds on JaVerT 2.0, taking the highly non-trivial step of designing the intermediate language, correctness results, and implementation to be parametric on the TL memory models. Despite this generalisation, Gillian substantially outperforms JaVerT 2.0, both for symbolic testing [19] and for verification.

Verifast [26] and the tool in [7] are prominent examples of semi-automatic tools that provide functionally-correct verification of C programs using separation-logic specifications. These tools work with C fragments and simplified memory models. While the tool in [7] has not been applied to real-world code, Verifast has been used to verify, e.g., an implementation of a Policy Enforcement Point (PEP) for Network Admission Control scenarios [38]. One difference between these tools and Gillian is that Gillian specifications can express negative resource, allowing us to differentiate missing resource errors from use-after-free errors. However, Verifast, unlike Gillian, supports reasoning about concurrent programs. There is also much work on using theorem provers to verify both sequential and concurrent C code using separation logic: see, for example, the DeepSpec project [45] and the Iris project [47], which we do not describe here.

**Related Formal C Memory Models.** Our compositional C memory models were inspired by CompCert [32] and the CH20 formalisation of Krebbers [29]. In particular, our concrete C model is adapted from the complete model of CompCert, which supports reasoning about programs that access in-memory data representations. This feature is used by the AWS deserialisation algorithm, which reads the buffer contents at the byte-granularity.

We present our compositional symbolic C memory model in this paper as a simple lifting of the concrete one. Our implementation is more complex, however, representing blocks as trees holding symbolic values and combining the concepts of memory trees and abstract values from the concrete memory model of the CH2O formalisation. Although not mentioned in [29], CH2O does keep track of some negative resource in that it maintains freed locations, but not block bounds.

**Analysis of the AWS Encryption SDK.** Amazon has recently directed considerable effort towards the formal analyses of their codebase, with a number of tools incorporated into their CI pipeline. For example, the main cryptographic algorithms of the AWS Encryption SDK have certified implementations in the specification language Cryptol [17], underpinned by SAW [12]. These implementations, however, have not yet been proven equivalent to the corresponding C implementation. In addition, the C implementation of the AWS Encryption SDK includes a symbolic test suite run using CBMC [30]. This implementation makes heavy use of the aws-c-common data-structure library, which is annotated with first-order assertions checked by CBMC. CBMC is a mature, industrial-strength tool, likely to outperform and have broader coverage than the symbolic testing of Gillian-C, with substantially fewer annotations than Gillian verification. However, as CBMC is a bounded model checker, it provides weaker correctness guarantees and is not compositional. Its expressivity is also somewhat constrained by the expressivity of the C runtime. For example, it does not allow reasoning about the size of allocated memory. Gillian specifications have this expressivity, as highlighted by the discovered over-allocation bug. The subtle logical bug found by Gillian also demonstrates the importance of being able to express full, functionally-correct specifications. We believe there has been no previous analysis of the JS implementation of AWS Encryption SDK.

## 7 Conclusions

We have introduced compositional verification to the Gillian platform. Our work includes a methodology for designing compositional TL memory models, distinguishing negative resource from missing resource and using the JS and C memory models as demonstrator examples. It also includes a novel, parametric approach to assertion interpretation, independent of the TL, enabling compositional use of function specifications in verification. We have been able to push the Gillian verification to self-contained, critical, real-world AWS JS and C code. The bugs and suggestions for code improvements that arose during this verification process have all been accepted by the developers and incorporated into the codebase. To our knowledge, this is the first time that industry-grade JS code has been fully verified and the first time that, in one verification platform, the same abstractions were used to verify industry code from languages as different as JS and C. The artifact accompanying this paper can be found at [34], and the entire Gillian development at [46]. In future, we will publish correctness results for Gillian verification [33], as part of an in-depth theoretical study of program correctness and incorrectness for symbolic testing, verification and bi-abductive reasoning being developed in Gillian.

**Acknowledgments.** We thank AWS engineers and Mike Dodds from Galois for several inspiring meetings which led to our focus on verifying the JS implementation of the AWS Encryption SDK message header deserialisation module. We would especially like to thank Ryan Emery for many detailed discussions about his JS code. We thank the reviewers, whose comments have improved the overall quality of the paper. Gardner and Maksimović were supported by the EPSRC Fellowship 'VetSpec: Verified Trustworthy Software Specification' (EP/R034567/1). Fragoso Santos was supported by national funds through Fundação para a Ciência e a Tecnologia (UIDB/50021/2020, INESC-ID multi-annual funding) and project INFOCOS (PTDC/CCI-COM/32378/2017). Ayoun was supported by a Department of Computing PhD Scholarship from Imperial.

## References

1. Amazon Web Services: AWS Encryption SDK: C Implementation. https://github.com/aws/aws-encryption-sdk-c (2020)
2. Amazon Web Services: AWS Encryption SDK: JS Implementation. https://github.com/aws/aws-encryption-sdk-javascript (2020)

3. Amazon Web Services: AWS Encryption SDK: Message Format. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/message-format.html (2020)
4. Appel, A.W., Blazy, S.: Separation Logic for Small-Step Cminor. In: TPHOL (2007)
5. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO (2005)
6. Botinčan, M., Distefano, D., Dodds, M., Grigore, R., Naudžiūnienė, D., Parkinson, M.J.: coreStar: The core of jstar. In: Boogie (2011)
7. Botinčan, M., Parkinson, M.J., Schulte, W.: Separation Logic Verification of C Programs with an SMT Solver. Electr. Notes Theor. Comput. Sci. **254**, 5–23 (2009)
8. Calcagno, C., Distefano, D.: Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: NFM (2011)
9. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving Fast with Software Verification. In: NFM (2015)
10. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. JACM **58**, 26:1–26:66 (2011)
11. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS (2007)
12. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous Formal Verification of Amazon s2n. In: CAV (2018)
13. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOL (2009)
14. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP (2010)
15. Distefano, D., Parkinson, M.: jStar: Towards practical verification for Java. In: OOPSLA (2008)
16. ECMA TC39: Test262 Test Suite. https://github.com/tc39/test262 (2017)
17. Erkök, L., Matthews, J.: High Assurance Programming in Cryptol. In: CSIIRW (2009)
18. Fournet, C., Swamy, N., Chen, J., Dagand, P.E., Strub, P.Y., Livshits, B.: Fully Abstract Compilation to JavaScript. In: POPL (2013)
19. Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.: Gillian, Part I: A Multi-language Platform for Symbolic Execution. In: PLDI (2020)
20. Fragoso Santos, J., Maksimović, P., Grohens, T., Dolby, J., Gardner, P.: Symbolic Execution for JavaScript. In: PPDP (2018)
21. Fragoso Santos, J., Maksimović, P., Naudžiūnienė, D., Wood, T., Gardner, P.: JaVerT: JavaScript Verification Toolchain. PACMPL **2**(POPL), 50:1–50:33 (2018)
22. Fragoso Santos, J., Maksimović, P., Sampaio, G., Gardner, P.: JaVerT 2.0: Compositional Symbolic Execution for JavaScript. PACMPL **3**(POPL), 66:1–66:31 (2019)
23. Frumin, D., Gondelman, L., Krebbers, R.: Semi-automated Reasoning About Non-determinism in C Expressions. In: PLS (2019)
24. Gardner, P., Maffeis, S., Smith, G.D.: Towards a Program Logic for JavaScript. In: POPL (2012)
25. Ishtiaq, S.S., O'Hearn, P.W.: BI as an Assertion Language for Mutable Data Structures. In: Hankin, C., Schmidt, D. (eds.) POPL (2001)

26. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: NFM (2011)
27. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28** (2018)
28. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects of Computing **27**(3), 573–609 (2015)
29. Krebbers, R.: A Formal C Memory Model for Separation Logic. Journal of Automated Reasoning **57**(4), 319–387 (2016)
30. Kroening, D., Tautschnig, M.: CBMC: C bounded model checker. In: TACAS (2014)
31. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (2012)
32. Leroy, X.: Formal Verification of a Realistic Compiler. Commun. ACM **52**(7), 107–115 (2009)
33. Maksimović, P., Santos, J.F., Ayoun, S.E., Gardner, P.: Gillian: A Multi-Language Platform for Unified Symbolic Analysis (2021), `http://arxiv.org/abs/2105.14769`
34. Maksimović, P., Ayoun, S.E., Fragoso Santos, J., Gardner, P.: Artifact: Gillian, Part II: Real-World Verification for JavaScript and C (2021), `https://doi.org/10.5281/zenodo.4838116`
35. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: VMCAI (2016)
36. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Dependable Software Systems Engineering (2017)
37. Nguyen, H.H., Kuncak, V., Chin, W.N.: Runtime checking for separation logic. In: VMCAI. pp. 203–217 (2008)
38. Philippaerts, P., Mülberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software Verification with VeriFast: Industrial Case Studies. Science of Computer Programming **82**, 77–97 (2014)
39. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In: CAV (2020)
40. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS (2002)
41. S. Panić: Collections-C: A Library of Generic Data Structures. `https://github.com/srdja/Collections-C` (2014)
42. Sampaio, G., Santos, J.F., Maksimović, P., Gardner, P.: A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications. In: ECOOP (2020)
43. Santos, M.: Buckets-js: A javascript data structure library. `https://github.com/mauriciosantos/Buckets-JS` (2016)
44. Swamy, N., Weinberger, J., Schlesinger, C., Chen, J., Livshits, B.: Verifying Higher-order Programs with the Dijkstra Monad. In: PLDI (2013)
45. The DeepSpec Team: The DeepSpec Project. `https://deepspec.org/main` (2021)
46. The Gillian Team: Gillian. `https://gillianplatform.github.io` (2020)
47. The Iris Team: The Iris Project. `https://iris-project.org` (2021)
48. Watt, C., Maksimović, P., Krishnaswami, N.R., Gardner, P.: A Program Logic for First-Order Encapsulated WebAssembly. In: ECOOP (2019)

# Debugging Network Reachability with Blocked Paths

S. Bayless(✉), J. Backes, D. DaCosta, B. F. Jones, N. Launchbury, P. Trentin, K. Jewell, S. Joshi, M. Q. Zeng, and N. Mathews

Amazon Web Services, Seattle, USA
sabayles@amazon.com

**Abstract.** In this industrial case study we describe a new network troubleshooting analysis used by VPC Reachability Analyzer, an SMT-based network reachability analysis and debugging tool. Our troubleshooting analysis uses a formal model of AWS Virtual Private Cloud (VPC) semantics to identify whether a destination is reachable from a source in a given VPC configuration. In the case where there is no feasible path, our analysis derives a *blocked path*: an infeasible but otherwise complete path that would be feasible if a corresponding set of VPC configuration settings were adjusted.

Our blocked path analysis differs from other academic and commercial offerings that either rely on packet probing (e.g., tcptrace) or provide only partial paths terminating at the first component that rejects the packet. By providing a complete (but infeasible) path from the source to destination, we identify for a user all the configuration settings they will need to alter to admit that path (instead of requiring them to repeatedly re-run the analysis after making partial changes). This allows users to refine their query so that the blocked path is aligned with their intended network behavior before making any changes to their VPC configuration.

## 1 Introduction

This paper describes a new network connectivity troubleshooting analysis used by VPC Reachability Analyzer, a service that analyzes Amazon Web Services' (AWS) Virtual Private Cloud (VPC) configurations.

VPCs are user-configured networks of virtual compute devices and resources. AWS VPC offers dozens of networking components and controls to give users flexibility in configuring their networks. Access to these resources is logically isolated within virtual networks configured by the users. As VPCs grow in size and complexity, users can increasingly benefit from automation to identify and resolve misconfigurations, as well as to validate that applications maintain security and availability invariants through infrastructure changes.

VPC Reachability Analyzer uses the Tiros [2] formal model of AWS VPC networking semantics to identify whether a destination is reachable from a source in a given VPC configuration. If the destination is reachable, then Tiros identifies a *feasible path* from the source to the destination, where a path is

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 851–862, 2021.
https://doi.org/10.1007/978-3-030-81688-9_39

a sequence of network components associated with incoming and/or outgoing packet header assignments (protocol, addresses, ports). The outgoing packet header of one component is the incoming packet header of the next component. Paths may also identify relevant VPC configuration details such as the specific routes, firewall rules, or other settings admitting the packet at each step. Each component in a VPC may accept or reject incoming and outgoing packet headers; a *feasible path* is a path in which every component on the path accepts both its incoming and outgoing packet header.

Tiros's analysis is static, *i.e.*, Tiros does not inject traffic into VPC configurations, and is complete for the subset of AWS VPC semantics it supports: if there exists a path connecting the source and destination, Tiros will find it. Since 2018, Tiros has powered the commercially available *Network Reachability* assessment in Amazon Inspector [1], statically identifying ports on EC2 Instances (virtual machines) accessible outside of their VPCs.

In this work, we extend Tiros by introducing a new diagnostic *blocked path* analysis when there is not a feasible path, to help users understand why their query is infeasible. A *blocked path* is a path as defined above, in which at least one component rejects its incoming or outgoing packet, along with one or more *blocking reasons*: elements of the VPC configuration preventing one or more components on the path from accepting packets. The blocked path identifies a sufficient set of blocking reasons, such that if each were addressed the query would be satisfiable.

Previous tools for connectivity diagnosis typically provide a partial path, up to the first component/rule that rejects the packet; in some cases those tools also identify a single blocking reason. Remediations based on a partial path may address that initial blocking reason only to discover that remediations are still necessary, or that the remediation may be working towards a path that the user ultimately will reject. Providing a complete blocked path connecting the source and destination allows users to ensure that their intent is aligned with our diagnosis before taking any corrective actions.

Our contributions in this work are:

1. Identifying the notion of a blocked path as a useful medium for conveying a network diagnosis and aligning it with a user's intent,
2. Demonstrating how blocked paths can be efficiently derived at scale,
3. Describing VPC Reachability Analyzer, a commercial tool based on these insights.

## 2 Background

### 2.1 Related Works

Many previous works have proposed network reachability diagnosis tools, including both widely-used industry tools and academic literature. These tools can be broadly divided into model-based and non-model-based approaches.

Non-model-based network diagnostic tools include system applications such as IPTRACE and TCPTRACE, commercial tools such as *Cisco Packet Tracer* [7], and academic works such as *Tulip* [12]. These tools trace live packets through a network or routing device, identifying the sequence of addresses of devices that accept the packet. Packet tracing tools lack visibility into the configuration settings that block and route packets.

Model-based tools [2,5,6,13,16] statically analyze reachability between a specified source and destination in a network or routing device. Rather than transmitting live packets, these tools use formal methods such as constraint solvers to rigorously identify feasible paths. Existing model-based tools provide control-plane level information when there is a feasible path, but produce either no information for unreachable paths, or identify only the first (out of potentially many) reasons why a path is blocked.

Our blocked path analysis is based on deriving minimal correction subsets (described below), which several previous works have proposed for general-purpose SAT-based error diagnosis or repair [4,8,9,17].

## 2.2 Minimal Correction Subsets

The blocked path analysis we describe in Sect. 3 relies on two related concepts: Maximal Satisfiable Subsets (MSS) and Minimal Correction Subsets (MCS), which we define below. Following the definitions from [14]:

**Definition 1 (MSS).** *$\mathcal{S} \subseteq \mathcal{F}$ is a Maximal Satisfiable Subset of constraints $\mathcal{F}$ iff $\mathcal{S}$ is satisfiable and $\forall c \in \mathcal{F} \setminus \mathcal{S}, \mathcal{S} \cup \{c\}$ is unsatisfiable.*

**Definition 2 (MCS).** *$\mathcal{C} \subseteq \mathcal{F}$ is a Minimal Correction Subset of constraints $\mathcal{F}$ iff $\mathcal{F} \setminus \mathcal{C}$ is satisfiable and $\forall c \in \mathcal{C}, (\mathcal{F} \setminus \mathcal{C}) \cup \{c\}$ is unsatisfiable.*

The complement of an MCS, $\mathcal{F} \setminus MCS(\mathcal{F})$, is guaranteed to be a maximal satisfiable subset of $\mathcal{F}$; for this reason the MCS is sometimes called the coMSS.[1]

In general, the MCS and MSS are not guaranteed to be unique. There is a close connection between the definition of a Maximal Satisfiable Subset and MaxSAT [10]: The largest MSS (and therefore smallest MCS) corresponds to a solution to MaxSAT. Indeed, one approach for computing the MCS is to compute MaxSAT and take the complement. Efficient algorithms for directly computing the (not necessarily smallest) MCS without computing MaxSAT are available and are typically much faster than computing MaxSAT; a good survey of MCS algorithms including an empirical evaluation can be found in [14].

In constraint optimization problems, it is common to consider hard and soft constraints, in which only the soft constraints may be relaxed. Definition 2 assumes that all constraints are soft, but can be easily extended to support

[1] Note that a minimal correction subset is a distinct concept from an unsatisfiable core [11]. An unsatisfiable core is always unsatisfiable, but its complement $\mathcal{F} \setminus CORE(\mathcal{F})$ is not guaranteed to be satisfiable; in contrast, an MCS may or may not be satisfiable, but its complement is guaranteed to be satisfiable.

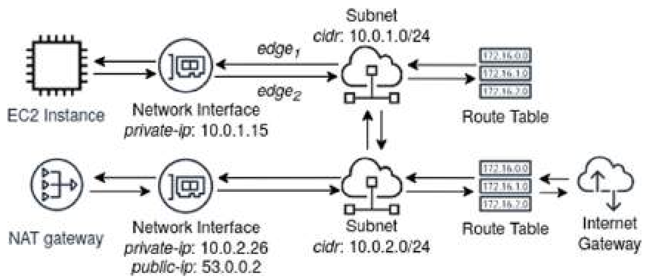
a mix of soft and hard constraints (where the MCS must contain only soft constraints). In this case, the MCS is only well defined if the hard constraints are satisfiable.

In Sect. 4, we will use a function COMPUTEMCS(*Soft*, *Hard*) that supports both hard and soft constraints. COMPUTEMCS returns a minimal correction set $\mathcal{C} = MCS(\mathit{Soft} \cup \mathit{Hard})$, with $\mathcal{C} \subseteq \mathit{Soft}$. Our implementation of COMPUTEMCS uses a simple binary search, similar to FastDiag [4], or Algorithm BFD from [14]. We add activation literals to the soft constraints to allow the underlying solver instance to be re-used incrementally while testing different subsets of soft constraints for satisfiability.

## 2.3 Network Reachability

We use the SMT-encoding of AWS VPC network semantics previously described in TIROS [2]. In this section, we briefly review this graph-based encoding; we refer readers to [2] for more details.

We take as input a configuration describing one or more user VPCs, and a user-specified reachability query, consisting of a source and destination component in the VPC. For example, the source of the query may be an internet gateway, and the destination may be an EC2 Instance. A query may also optionally specify additional constraints, such as the protocol, a range of source or destination addresses or ports for the packet, or an intermediate component that must (or must not) be on the path.



| Packet Header | |
|---|---|
| protocol | bv:8 |
| srcAdr | bv:32 |
| dstAdr | bv:32 |
| srcPort | bv:16 |
| dstPort | bv:16 |

$$((dstAdr \neq 10.0.1.15) \implies \neg edge_1)$$
$$((srcAdr \neq 10.0.1.15) \implies \neg edge_2)$$

**Fig. 1.** Simplified example symbolic graph representation of a VPC (*left*), with symbolic packet header consisting of bitvectors (*right*). Edges in the graph are associated with theory atoms, and are traversable only if those atoms are assigned true. Two example constraints, enforcing that a network interface is only accessible if the packet is addressed to/from that interface are shown. These constraints relate edge atoms in the symbolic graph to the bitvectors in the symbolic packet header to enforce AWS VPC semantics.

We encode VPC configurations as constrained symbolic graphs using the SMT solver MonoSAT [3], with fixed-width bitvectors representing the protocol, port, and addressing information in a symbolic packet header. Figure 1 shows a symbolic graph along with a packet header and example constraints.

VPC components are represented as a nodes in the symbolic graph. Each component has semantics governing which packets it will accept; these semantics are encoded as constraints that restrict which edges incident to that component's node are traversible, depending on the assignment of the packet header variables. A satisfying assignment to the full set of constraints corresponds to a feasible path. In such an assignment, the bitvector variable assignments provide the packet header(s) and the graph theory model provides a path of network component nodes connecting the source and destination of the user's query.

Some components (such as NAT gateways) transform and retransmit packets. Tiros supports this by unrolling the VPC configuration graph into multiple copies with separate packet header variables. Edges from packet-transforming components connect to their components in the next unrolled section of the graph. Tiros unrolls the graph to a sufficient depth to model the behavior of the components for each query.

Query source and destination reachability is enforced with a single graph theory reachability predicate requiring a feasible path in the VPC configuration graph from the source to the destination of the query. Query restrictions requiring intermediate components are enforced using additional reachability predicates. Query restrictions requiring that a given resource not occur on a path are enforced by excluding that resource from the VPC configuration graph representation. Packet header restrictions are enforced using bitvector constraints.

If the constraints are satisfiable, Tiros extracts a reachable path satisfying the query from the satisfying assignment to the constraints. In the next section, we will discuss how we extend Tiros to also provide diagnostic feedback in the case where the constraints are unsatisfiable.

## 3 Blocked Paths for Network Configuration Diagnosis

We introduce the notion of *blocked path* for analyzing infeasible network connections. As shown in Fig. 2, a blocked path is an infeasible but otherwise complete path from a source to a destination, in which one or more edges or nodes are annotated with *blocking reasons*: configuration settings or network semantics that explain why that transition in the path is infeasible.

Unlike a live packet trace, a blocked path continues past components that reject or redirect the packet so as to reach the user's intended destination, potentially transiting through multiple infeasible steps along the way.

**Definition 3 (Blocked path).**

1. *A blocked path is a complete (but infeasible) path from a source to a destination in a network, satisfying the user's query.*

2. *A blocked path is* actionable*: it is a path that could, with the right control plane configuration adjustments, be a feasible path.*
3. *A blocked path identifies a sufficient set of* blocking reasons *(network semantics or control-plane settings) that would need to be addressed to admit the packet along that blocked path. This may include multiple blocking reasons along the path, as opposed to just the first blocking reason.*

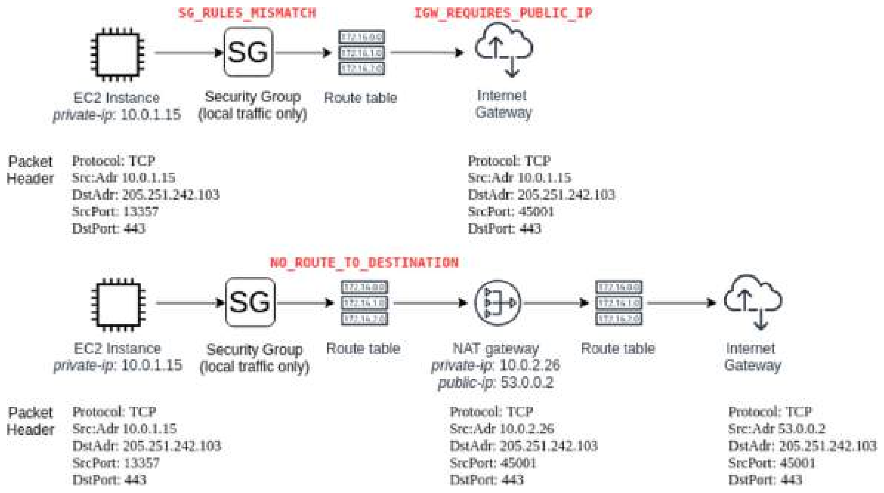


**Fig. 2.** Two alternative blocked paths from an EC2 instance to an internet gateway. These blocked paths take different routes, and have different *blocking reasons* (shown in red) that explain why those paths are infeasible. In the first blocked path, there are two blocking reasons: the security group egress rule rejects packets destined for the Internet, and the internet gateway requires that the source instance must have a public IP address. Note that although the packet would be rejected by the security group, the blocked path continues past the security group to identify a complete (but infeasible) path to the internet gateway. The second blocked path transitions through an intermediate NAT gateway, which satisfies the security group rule and also has a public IP address. However, this path is still blocked, because the route table does not have an applicable route to the NAT gateway.

## Validating User Intent

Showing a complete path from the source to destination, along with all the relevant configuration settings blocking that path, allows users to confirm that this course of action matches their intended network behavior before making any changes. However, in many cases there are multiple ways to adjust a configuration to admit a path, resulting in different blocked paths.

For example, Fig. 2 shows two example blocked paths to an internet gateway from an EC2 instance lacking a public IP address. Our analysis might initially produce for the user the shorter blocked path. Two remediation steps are required to admit this shorter path: The user must adjust the security group rule of the instance to admit egress packets to the public internet, and the user must also associate a public IP address with the source instance. Upon seeing the complete blocked path, the user may immediately determine that this would be the wrong solution for their network.

If the proposed blocked path doesn't match the user's intent, we allow users to submit a refined query so as to generate an alternative blocked path. For instance, the user may specify allowed address or port ranges for the packet, or specify components that must or must not appear on the path. Similarly, the user may submit a refined query specifying that a NAT gateway must be an intermediate component on the path. In this case, we might produce the longer blocked path from Fig. 2.

**Actionable Blocked Paths**

In some cases, there may not exist any combination of VPC configuration adjustments that would allow a query to be satisfied. For example, under typical conditions in VPCs, route tables cannot be adjusted to redirect packets that are destined for a local address within the VPC. It is possible for users to specify queries that cannot be satisfied without violating this local route restriction.

In principle, it is possible to derive a blocked path with non-user-configurable blocking reasons, however the resulting paths may behave in misleading or confusing ways, and in general will not be possible for users to actually achieve in any real configuration of their VPC. If possible, we want to ensure that the path contains only user-configurable blocking reasons, so that we produce an *actionable* finding for users. However, we still want to be able to provide useful diagnostics in cases where no actionable blocked path is possible (*e.g.*, to explain to the user that the local route restriction will prevent their path).

In Sect. 4, we describe how we determine when it is not possible to produce a blocked path without including non-configurable blocking reasons. In this case, we produce a partial path up to that first non-configurable blocking reason.

Additionally, in some cases a user may specify a query that remains unsatisfiable even if all of the network semantics in our model are relaxed. This can occur if the user specifies components that do not exist, or that are in isolated, disconnected networks (for which no relaxation of the edge constraints will admit a path). In this case, our blocked path analysis fails, and TIROS falls back on other techniques to produce diagnostic information.

In Sect. 5 we show that in most cases, our analysis succeeds and produces an actionable blocked path.

## 4 Deriving Blocked Paths from Unsatisfiable Queries

We group VPC configuration semantics into three disjoint sets of constraints: $(U \cup N \cup H)$. Set $U$ contains constraints that enforce user-configurable control-plane settings (such as a user-defined route or firewall rule), while set $N$ contains non-configurable for user-visible network semantics (such as the local route restriction).

Set $H$ contains elements of the constraints that are either not user-visible (such as internal implementation details) or that should never be relaxed (such as the reachability predicate or any other constraints defined by the user's query). For example, many of our constraints involve containment comparisons between CIDRs and bitvectors representing IP addresses. An individual CIDR comparison is encoded as a fresh literal represnting the truth value of the comparison, along with multiple clauses that enforce the comparison semantics. The intermediate clauses that enforce the comparison semantics are implementation details that we include in set $H$, ensuring they are not included in the blocking reasons.

When a query is unsatisfiable, we derive a blocked path and corresponding blocking reasons from a Maximal Satisfiable Subset and Minimal Correction Subset of $(U \cup N \cup H)$, with set $H$ being treated as hard constraints that must not be included in the MCS.

If possible, we want to produce an MCS containing only configurable blocking reasons from $U$. This ensures that the resulting blocked path is actionable. If we directly compute the MCS of the full constraint set $U \cup N \cup H$, with both $U$ and $N$ as soft constraints, non-configurable constraints from $N$ may be included in the MCS even in cases where there exists an MCS containing only constraints from $U$. On the other hand, we still want to be able to produce an MCS in the case where the non-configurable and hard constraints $(N \cup H)$ are, by themselves, unsatisfiable.

In Algorithm 1, we resolve this by breaking the computation of the MCS into two steps, initially computing an MCS of $N \cup H$, and only allowing constraints from $N$ into the blocking reasons if MCS$(N \cup H)$ is non-empty.

When $N \cup H$ is satisfiable, Algorithm 1 produces a blocked path that only contains the configurable blocking reasons from $U$.

Algorithm 1 constructs two correction sets, $MCS_N \subseteq N$ and $MCS_U \subseteq U$, with $MCS_N \cup MCS_U$ a valid MCS of $(U \cup N \cup H)$. We then extract a path $p$ from a satisfying assignment to the corresponding MSS $(U \cup N \cup H) \setminus (MCS_N \cup MCS_U)$. Finally, as shown below, we return either a complete or a partial blocked path, by associating blocking reasons from the MCS with nodes on that path.

Algorithm 1 relies on two helper methods, ExtractPath and BuildPath. ExtractPath retrieves the satisfying theory model (a sequence of edges) for the query reachability predicate from a satifiable formula, using the graph theory in the SMT-solver MonoSAT, and associates packet header assignments with each step of that path from the corresponding bitvector assignments. BuildPath maps the literals of the MCS to descriptive strings representing blocking reasons, and associates those strings with steps on the blocked path.

**Algorithm 1.** Blocked Path Analysis

```
1: function DeriveBlockedPath($U, N, H$)    ▷ Precondition: $U \cup N \cup H$ is UNSAT.
2:     if UNSAT($H$) then
3:         throw Error: No blocked path can be produced.
4:     else
5:         // Note: If $N \cup H$ is SAT, then $MCS_N = \emptyset$.
6:         $MCS_N \leftarrow$ computeMCS($N, H$)
7:         // Note: $(N \cup H) \setminus MCS_N$ is SAT; $MCS_U$ is well-defined.
8:         $MCS_U \leftarrow$ computeMCS($U, (N \cup H) \setminus MCS_N$)
9:         $p \leftarrow$ ExtractPath($(U \cup N \cup H) \setminus (MCS_N \cup MCS_U)$)
10:        return BuildPath($p, MCS_N, MCS_U$)
11:    end if
12: end function
```

We can see that $MCS_N \cup MCS_U$ meets the definition of a minimal correction set of $U \cup N \cup H$ by observing that:

$$\begin{aligned}
\text{SAT}((U \cup N \cup H \setminus (MCS_N)) \setminus (MCS_N \cup MCS_U)) &\quad \text{line 8}\\
\implies \text{SAT}((U \cup N \cup H) \setminus (MCS_N \cup MCS_U))) &\\
\forall c \in MCS_N, \text{UNSAT}((N \cup H) \setminus (MCS_N \setminus \{c\}) &\quad \text{line 6}\\
\forall c \in MCS_U, \text{UNSAT}((U \cup N \cup H) \setminus (MCS_U \setminus \{c\})) &\quad \text{line 8}\\
\implies \forall c \in (MCS_N \cup MCS_U), \text{UNSAT}((U \cup N \cup H) \setminus ((MCS_N \cup MCS_U) \setminus \{c\})) &
\end{aligned}$$

If $N \cup H$ is satisfiable, then $MCS_N$ is empty and $MCS_U$, containing only configurable constraints, is an MCS of $(U \cup N \cup H)$. In this case, BuildPath constructs a complete blocked path consisting entirely of configurable blocking reasons.

If $N \cup H$ is unsatisfiable, then $MCS_N$ is non-empty and $MCS_N \cup MCS_U$ contains at least one non-actionable constraints. In this case, the path $p$ may behave unexpectedly and may not be realizable in a VPC configuration after adjustment. If $MCS_N$ is non-empty, BuildPath forms the blocked path as above, but returns only the prefix of that blocked path up to and including the first edge or node associated with a non-actionable setting.

Above, we discussed the cases where $N \cup H$ is satisfiable or unsatisfiable. There is also a third possibility: The hard constraints $H$, representing the constraints enforcing the user's query or implementation details of our model, may by themselves be unsatisfiable. For example, $H$ may be unsatisfiable if the user specifies a source and destination that are in separate, disconnected networks.

If $H$ is unsatisfiable, Algorithm 1 fails, and is unable to produce even a partial blocked path. In this case, we fall back on other techniques to provide useful diagnostic information for users. In practice, the typical reason that $H$ is unsatisfiable is that the source and destination are in disconnected VPCs (so the reachability constraint is unsatisfiable). We use a static analysis pass to identify this case and handle it separately in our service.

In the case that Algorithm 1 produces a complete (*resp.* partial) blocked path, the underlying MCS algorithm guarantees that the blocked path will have the fewest possible number of blocking reasons from among all complete (*resp.* partial) blocked paths. In general this blocked path is not unique.

In our implementation of Algorithm 1, the graph-based decision heuristic in MonoSAT will prioritize finding shortest-length paths in most cases, but does not guarantee that a shortest-length path is always found.

## 5 Evaluation

VPC Reachability Analyzer, a commercial offering available from AWS since December 2020, uses the blocked path analysis we have described to derive findings for queries between unreachable endpoints.

To demonstrate the practical impact of this blocked path analysis, we randomly selected 1000 unreachable queries processed by VPC Reachability Analyzer. We executed the blocked path analysis for those queries on an 'm5.24xlarge' EC2 instance using GNU Parallel [15], running Amazon Linux 2, using MonoSAT version 1.6.0.

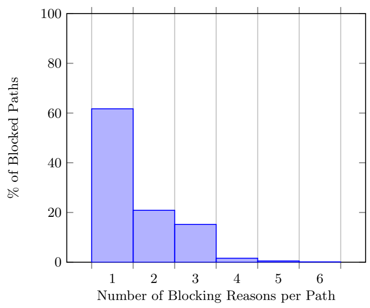


**Fig. 3.** Number of blocking reasons per blocked path (among the 63% of unreachable queries for which the blocked path analysis produced a complete blocked path). 97% percent of blocked paths have three or fewer blocking reasons; 60% have just a single blocking reason.

Excluding the time to complete the blocked path analysis, the average time required to initially determine satisfiability of the constraints was 2.1 s (P50: 1.7 s, P99: 7.4 s). The blocked path analysis was as fast or faster than the initial solving time, requiring 0.3 s on average (P50: 0.05 s, P99: 6.6 s).

As described in Sect. 4, in some cases, the blocked path analysis can produce only a partial path, or no results at all. Of those 1000 unreachable queries, 63.2% resulted in complete blocked paths, 7.4% resulted in partial blocked paths, and the remainder (29.4%) produced no analysis (in which case VPC Reachability Analyzer applies other techniques so that it can still provide useful diagnostics).[2]

As can be seen in Fig. 3, most blocked paths have just one blocking reason, and 97% have at most three. This demonstrates that our analysis produces actionable, concise findings on real production data, a key requirement of a useful diagnosis service.

## 6 Conclusion

The blocked path analysis we have introduced provides key advantages over previous network diagnostic techniques. By showing users a blocked path from a source to a destination, we allows users the opportunity to refine their query such that their intended path is aligned with our analysis. Furthermore, showing all blocking reasons on a blocked path allows users to understand the VPC configuration adjustments necessary to realize a path for their query.

Our blocked path analysis is a fully static analysis (requiring no packets to be injected into the network), can be computed efficiently using standard techniques from the formal methods literature, and is now used successfully in production by VPC Reachability Analyzer.

## References

1. Amazon Inspector. https://docs.aws.amazon.com/inspector/. Accessed December 2018
2. Backes, J., et al.: Reachability analysis for AWS-based networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 231–241. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_14
3. Bayless, S., Bayless, N., Hoos, H.H., Hu, A.J.: SAT modulo monotonic theories. In: Proceedings of AAAI, pp. 3702–3709 (2015)
4. Felfernig, A., Schubert, M., Zehentner, C.: An efficient diagnosis algorithm for inconsistent constraint sets. Artif. Intell. Eng. Design Anal. Manuf. AI EDAM **26**(1), 53 (2012)
5. Fogel, A., et al.: A general approach to network configuration analysis. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation. pp. 469–483. NSDI 2015, USENIX Association, Berkeley, CA, USA (2015). http://dl.acm.org/citation.cfm?id=2789770.2789803
6. Jayaraman, K., Bjørner, N., Outhred, G., Kaufman, C.: Automated analysis and debugging of network connectivity policies. Microsoft Research, pp. 1–11 (2014)
7. Jazib Frahim, Omar Santos, A.O.: Cisco ASA All-in-One Firewall, IPS, and VPN Adaptive Security Appliance, 3rd edition. Cisco Press (2014)

[2] Of the queries for which no blocked path analysis was performed, 80% were due to users specifying endpoints in disconnected VPCs. We perform a disconnected component analysis to identify this case. Others were due to users specifying resources they lack access to, or that we do not support.

8. Junker, U.: Preferred explanations and relaxations for over-constrained problems. In: AAAI-2004 (2004)
9. Koitz, R., Wotawa, F.: Sat-based abductive diagnosis. In: DX@ Safeprocess, pp. 167–176 (2015)
10. Li, C.M., Manya, F.: Maxsat, hard and soft constraints. Handb. Satisf. **185**, 613–631 (2009)
11. Lynce, I., Marques-Silva, J.P.: On computing minimum unsatisfiable cores (2004)
12. Mahajan, R., Spring, N., Wetherall, D., Anderson, T.: User-level internet path diagnosis. ACM SIGOPS Oper. Syst. Rev. **37**(5), 106–119 (2003)
13. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, B., King, S.T.: Debugging the data plane with anteater. In: Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, 15–19 August 2011, pp. 290–301 (2011). https://doi.org/10.1145/2018436.2018470, http://doi.acm.org/10.1145/2018436.2018470
14. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: Twenty-Third International Joint Conference on Artificial Intelligence. Citeseer (2013)
15. Tange, O.: GNU Parallel 2018. Ole Tange, March 2018. https://doi.org/10.5281/zenodo.1146014
16. Tian, B., et al.: Safely and automatically updating in-network acl configurations with intent language. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 214–226 (2019)
17. Walter, R., Felfernig, A., Küchlin, W.: Constraint-based and sat-based diagnosis of automotive configuration problems. J. Intell. Inf. Syst. **49**(1), 87–118 (2017)

# Lower-Bound Synthesis Using Loop Specialization and Max-SMT

Elvira Albert[1,2], Samir Genaim[1,2], Enrique Martin-Martin[1], Alicia Merayo[1](✉), and Albert Rubio[1,2]

[1] Fac. Informática, Complutense University of Madrid, Madrid, Spain
amerayo@ucm.es

[2] Instituto de Tecnología del Conocimiento, Madrid, Spain

**Abstract.** This paper presents a new framework to synthesize lower-bounds on the worst-case cost for non-deterministic integer loops. As in previous approaches, the analysis searches for a *metering function* that under-approximates the number of loop iterations. The key novelty of our framework is the *specialization* of loops, which is achieved by restricting their enabled transitions to a subset of the inputs combined with the narrowing of their transition scopes. Specialization allows us to find metering functions for complex loops that could not be handled before or be more precise than previous approaches. Technically, it is performed (1) by using quasi-invariants while searching for the metering function, (2) by strengthening the loop guards, and (3) by narrowing the space of non-deterministic choices. We also propose a Max-SMT encoding that takes advantage of the use of soft constraints to force the solver look for more accurate solutions. We show our accuracy gains on benchmarks extracted from the 2020 Termination and Complexity Competition by comparing our results to those obtained by the LoAT system.

## 1 Introduction

One of the most important problems in program analysis is to automatically –and accurately– bound the cost of program's executions. The first automated analysis was developed in the 70s [24] for a strict functional language and, since then, a plethora of techniques has been introduced to handle the peculiarities of the different programming languages (see, e.g., for Integer programs [5], for Java-like languages [2,19], for concurrent and distributed languages [16], for probabilistic programs [15,18], etc.) and to increase their accuracy (see, e.g., [10,14,21,22]). The vast majority of these techniques have focused on inferring *upper bounds* on the worst-case cost, since having the assurance that none execution of the program will exceed the inferred amount of resources (e.g., time, memory, etc.) has crucial applications in safety-critical contexts. On the other hand, *lower bounds*

This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) project RTI2018-094403-B-C31, by the CM project S2018/TCS-4314 co-funded by EIE Funds of the European Union, and by the UCM CT42/18-CT43/18 grant.

on the best-case cost characterize the minimal cost of any program execution and are useful in task parallelization (see, e.g., [3,9,10]). There are a third type of important bounds which are the focus of this work: *lower bounds on the worst-case cost*, they bound the worst-case cost from below. Their main application is that, together with the upper bounds on worst-case, allow us to infer tighter worst-case cost bounds (when they coincide ensuring that the inferred cost is exact) what can be crucial in safety-critical contexts. Besides, lower bounds on the worst-case cost will give us families of inputs that lead to an expensive cost, what could be used to detect performance bugs. In what follows, we use the acronyms $\mathrm{LB}^w$ and $\mathrm{LB}^b$ to refer to *w*orst-case and *b*est-case lower-bounds, resp.

*State-of-the-Art in $LB^w$.* An important difference between $\mathrm{LB}^w$ and $\mathrm{LB}^b$ is that, while the best-case must consider *all* program runs, $\mathrm{LB}^w$ holds for (usually infinite) families of the most expensive program executions. This is why the techniques applicable to $\mathrm{LB}^b$ inference (e.g., [3,9,10]) are not useful for $\mathrm{LB}^w$ in general, since they would provide too inaccurate (low) results. The state-of-the-art in $\mathrm{LB}^w$ inference is [12,13] (implemented in the LoAT system) which introduces a variation of ranking functions, called *metering functions*, to under-estimate the number of iterations of *simple* loops, i.e., loops without branching nor nested loops. The core of this method is a simplification technique that allows treating general loops (with branchings and nested loops) by using the so-called *acceleration*: that replaces a transition representing one loop iteration by another rule that collects the effect of applying several consecutive loop iterations using the original rule. Asymptotic lower bounds are then deduced from the resulting simplified programs using a special-purpose calculus and an SMT encoding.

*Motivation.* Our work is motivated by the limitation of state-of-the-art methods when, by treating each simple loop separately, a $\mathrm{LB}^w$ bound cannot be found or it is too imprecise. For example, consider the interleaved loop in Fig. 1, that is a simplification of the benchmark SimpleMultiple.koat from the Termination and Complexity competition. Its *transition system* appears to the right (the transition system is like a control-flow graph (CFG) in which the transitions $\tau$ are labeled with the applicability conditions and with the updates for the variables, primed variables denote the updated values). In every iteration $x$ or $y$ can decrease by one, and these behaviors can interleave. The worst case is obtained for instance when $x$ is decreased to 0 ($x$ iterations) and then $y$ is decreased to 0 ($y$ iterations), resulting in $x + y$ iterations, or when $y$ is first decreased to 1 and then $x$ to $-1$, etc. The approach in [12,13] accelerates independently both $\tau_1$ and $\tau_4$, resulting in accelerated versions $\tau_1^a = x \geq -1 \wedge y > 0 \wedge x' = -1 \wedge y' = y$ with cost $x + 1$ and $\tau_4^a = x \geq 0 \wedge y \geq 0 \wedge x' = x \wedge y' = 0$ with cost $y$. Applying one accelerated version results in that the other accelerated version cannot be applied because of the final values of the variables. Thus, the overall knowledge extracted from the loop is that it can iterate $x+1$ or $y$ times, whereas the precise $\mathrm{LB}^w$ is $x+y$ iterations. Our challenge for inferring more precise $\mathrm{LB}^w$ is to devise a method that can handle all loop transitions simultaneously, as disconnecting them leads to a semantics loss that cannot be recovered by acceleration.

```
while (x >= 0 && y > 0) {
  if (*) {
    x = x - 1;
  } else {
    y = y - 1;
  }
}
```

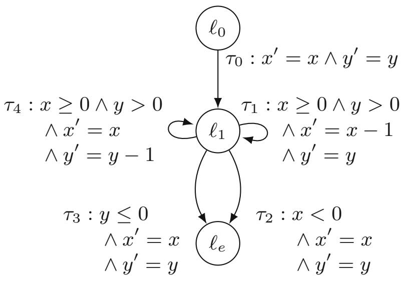

$\ell_0$
$\tau_0 : x' = x \wedge y' = y$
$\tau_4 : x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1$
$\ell_1$
$\tau_1 : x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y$
$\tau_3 : y \leq 0 \wedge x' = x \wedge y' = y$
$\ell_e$
$\tau_2 : x < 0 \wedge x' = x \wedge y' = y$

**Fig. 1.** Interleaved loop (left) and its representation as a transition system (right)

*Non-Termination and $LB^w$.* Our work is inspired by [17], which introduces the powerful concept of *quasi-invariant* to find witnesses for non-termination. A quasi-invariant is an invariant which does not necessarily hold on initialization, and can be found as in template-based verification [23]. Intuitively, when there is a loop in the program that can be mapped to a quasi-invariant that forbids executing any of the outgoing transitions of the loop, then the program is non-terminating. This paper leverages such powerful use of quasi-invariants and Max-SMT in non-termination analysis to the more difficult problem of LB$^w$ inference. Non-termination and LB$^w$ are indeed related properties: in both cases we need to find witnesses, resp., for non-terminating the loop and for executing at least a certain number of iterations. For LB$^w$, we additionally need to provide such under-estimation for the number of iterations and search for LB$^w$ behaviors that occur for a class of inputs rather than for a single input instantiation (since the LB$^w$ for a single input is a concrete (i.e., constant) cost, rather than a parametric LB$^w$ function as we are searching for). Instead, for non-termination, it is enough to find a non-terminating input instantiation.

*Our Approach.* A fundamental idea of our approach is to *specialize* loops in order to guide the search of the metering functions of complex loops, avoiding the inaccuracy introduced by disconnecting them into simple loops. To this purpose, we propose specializing loops by combining the addition of constraints to their transitions with the restriction of the valid states by means of quasi-invariants. For instance, for the loop in Fig. 1, our approach automatically narrows $\tau_1$ by adding $x > 0$ (so that $x$ is decreased until $x = 0$) and $\tau_4$ by adding $x \leq 0$ (so that $\tau_4$ can only be applied when $x = 0$). This specialized loop has lost many of the possible interleavings of the original loop but keeps the worst case execution of $x+y$ iterations. These specialized guards do not guarantee that the loop executes $x + y$ iterations in every possible state, as the loop will finish immediately for $x < 0$ or $y \leq 0$, thus our approach also infers the quasi-invariant $x \geq 0 \wedge x \leq y$. Combining the specialized guards and the quasi-invariant, we can assure that when reaching the loop in a valid state according to the quasi-invariant, $x + y$ is a lower bound on the number of iterations of the loop, i.e., its cost. Using quasi-invariants that include all (invariant) inequalities syntactically appearing in loop transitions might work for the case of loops with single path. However, for the general case, the specialized guards usually lead to essential quasi-invariants that do not appear in the original loop. The specialization achieved by adding constraints could be also applied in the context of non-termination to increase the accuracy of [17], as only quasi-invariants were used. Therefore, we argue that our work avoids the precision loss caused by the simplification in [12,13] and, besides, introduces a loop specialization technique that can also be applied to gain precision in non-termination analysis [17].

*Contributions.* Briefly, our main theoretical and practical contributions are:

1. In Sect. 3 we introduce several semantic specializations of loops that enable the inference of *local* metering functions for complex loops by: (1) restricting the input space by means of automatically generated quasi-invariants, (2) narrowing transition guards and (3) narrowing non-deterministic choices.
2. We propose a template-based method in Sect. 4 to automate our technique which is effectively implemented by means of a Max-SMT encoding. Whereas the use of templates is not new [6], our encoding has several novel aspects that are devised to produce better lower-bounds, e.g., the addition of (soft) constraints that force the solver look for larger lower-bound functions.
3. We implement our approach in the LOBER system and evaluate it on benchmarks from the Integer Transition Systems category of the 2020 Termination and Complexity Competition (see Sect. 5). Our experimental results when compared to the existing system LoAT [12] are promising: they show further accuracy of LOBER in challenging examples that contain complex loops.

## 2 Background

This section introduces some notation on the program representation and recalls the notion of LB$^w$ we aim at inferring.

### 2.1 Program Representation

Our technique is applicable to sequential non-deterministic programs with integer variables and commands whose updates can be expressed in linear (integer) arithmetic. We assume that the non-determinism originates from non-deterministic assignments of the form "x:=nondet();", where x is a program variable and nondet() can be represented by a fresh non-deterministic variable u. This assumption allows us to also cover non-deterministic branching, e.g., "if (*){..} else {..}" as it can be expressed by introducing a non-deterministic variable u and rewriting the code as "u:=nondet(); if (u≥0){..} else {..}".

Our programs are represented using *transition systems*, in particular using the formalization of [17] that simplifies the presentation of some formal aspects of our work. A transition system (abbrev. TS) is a tuple $\mathcal{S} = \langle \bar{x}, \bar{u}, \mathcal{L}, \mathcal{T}, \Theta \rangle$, where $\bar{x}$ is a tuple of $n$ integer program variables, $\bar{u}$ is a tuple of integer (non-deterministic) variables, $\mathcal{L}$ is a set of locations, $\mathcal{T}$ is a set of transitions, and $\Theta$ is a formula that defines the valid input and is specified by a conjunction of linear constraints of the form $\bar{a}\cdot\bar{x}+b\diamond 0$ where $\diamond \in \{>,<,=,\geq,\leq\}$. A transition is of the form $(\ell,\ell',\mathcal{R}) \in \mathcal{T}$ such that $\ell,\ell' \in \mathcal{L}$, and $\mathcal{R}$ is a formula over $\bar{x}$, $\bar{u}$ and $\bar{x}'$ that is specified by a conjunction of linear constraints of the form $\bar{a}\cdot\bar{x}+\bar{b}\cdot\bar{u}+\bar{c}\cdot\bar{x}'+d\diamond 0$ where $\diamond \in \{>,<,=,\geq,\leq\}$, and primed variables $\bar{x}'$ represent the values of the unprimed corresponding variables after the transition. We sometimes write $\mathcal{R}$ as $\mathcal{R}(\bar{x},\bar{u},\bar{x}')$, use $\mathcal{R}(\bar{x})$ to refer to the constraints that involve only variables $\bar{x}$ (i.e., the *guard*), and use $\mathcal{R}(\bar{x},\bar{u})$ to refer to the constraints that involve only variables $\bar{u}$ and (possibly) $\bar{x}$. W.l.o.g., we may assume that constraints involving primed variables are of the form $x_i' = \bar{a}\cdot\bar{x}+\bar{b}\cdot\bar{u}+c$. This is because non-determinism can be moved to $\mathcal{R}(\bar{x},\bar{u})$ – if a primed variable $x_i'$ appears in any expression that is not of this form, we replace $x_i'$ by a fresh non-deterministic variable $u_i$ in such expressions and add the equality $x_i' = u_i$. We require that for any $\bar{x}$ satisfying $\mathcal{R}(\bar{x})$, there are $\bar{u}$ satisfying $\mathcal{R}(\bar{x},\bar{u})$, formally

$$\forall\bar{x}.\exists\bar{u}.\ \mathcal{R}(\bar{x}) \rightarrow \mathcal{R}(\bar{x},\bar{u}) \tag{1}$$

This guarantees that for any state $\bar{x}$ satisfying the condition, there are values for the non-deterministic variables $\bar{u}$ such that we can make progress. A transition that does not satisfy this condition is called *invalid*. Note that (1) does not refer to $\bar{x}'$ since they are set in a deterministic way, once the values of $\bar{x}$ and $\bar{u}$ are fixed. W.l.o.g., we assume that all coefficients and free constants, in all linear constraints, are integer; and that there is a single *initial location* $\ell_0 \in \mathcal{L}$ with no incoming transitions, and a single *final location* $\ell_e$ with no outgoing transitions.

*Example 1.* The TS graphically presented in Fig. 1 is expressed as follows, considering that all inputs are valid ($\Theta = true$):

$$\begin{aligned}\mathcal{S} \equiv \langle\ & \{x,y\},\emptyset,\{\ell_0,\ell_1,\ell_e\},\\ & \{(\ell_0,\ell_1,x'=x\wedge y'=y),\\ & \ (\ell_1,\ell_1,x\geq 0\wedge y>0\wedge x'=x-1\wedge y'=y),\\ & \ (\ell_1,\ell_e,x<0\wedge x'=x\wedge y'=y),\\ & \ (\ell_1,\ell_e,y\leq 0\wedge x'=x\wedge y'=y),\\ & \ (\ell_1,\ell_1,x\geq 0\wedge y>0\wedge x'=x\wedge y'=y-1)\},\mathit{true}\rangle\end{aligned}$$

A configuration $C$ is a pair $(\ell,\sigma)$ where $\ell \in \mathcal{L}$ and $\sigma : \bar{x} \mapsto \mathbb{Z}$ is a mapping representing a state. We abuse notation and use $\sigma$ to refer to $\wedge_{i=1}^{n} x_i = \sigma(x_i)$, and also write $\sigma'$ for the assignment obtained from $\sigma$ by renaming the variables to primed variables. There is a transition from $(\ell,\sigma_1)$ to $(\ell',\sigma_2)$ iff there is $(\ell,\ell',\mathcal{R}) \in \mathcal{T}$ such that $\exists\bar{u}.\sigma_1 \wedge \sigma_2' \models \mathcal{R}$. A (valid) trace $t$ is a (possibly infinite) sequence of configurations $(\ell_0,\sigma_0),(\ell_1,\sigma_1),\ldots$ such that $\sigma_0 \models \Theta$, and for each $i$ there is a transition from $(\ell_i,\sigma_i)$ to $(\ell_{i+1},\sigma_{i+1})$. Traces that are infinite or end in a configuration with location $\ell_e$ are called complete. A configuration $(\ell,\sigma)$, where $\ell \neq \ell_e$, is *blocking* iff

$$\sigma \not\models \bigvee_{(\ell,\ell',\mathcal{R})\in\mathcal{T}} \mathcal{R}(\bar{x}) \tag{2}$$

A TS is non-blocking if no trace includes a blocking configuration. We assume that the TS under consideration is non-blocking, and thus any trace is a prefix of a complete one. Throughout the paper, we represent a TS as a CFG, and analyze its strongly connected components (SCC) one by one. An SCC is said to be *trivial* if it has no edge.

## 2.2 Lower-Bounds

For simplicity, we assume that an execution step (a transition) costs 1. Under this assumption, the cost of a trace $t$ is simply its length $len(t)$ where the length of an infinite trace is $\infty$. In what follows, the set of all configurations is denoted by $\mathcal{C}$, the set of all valid complete traces (using a transition system $\mathcal{S}$) when starting from configuration $C \in \mathcal{C}$ is denoted by $Traces_{\mathcal{S}}(C)$, and $\mathbb{R}_{\geq 0} = \{k \in \mathbb{R} \mid k \geq 0\} \cup \{\infty\}$. For a non-empty set $M \subseteq \mathbb{R}_{\geq 0}$, $sup\ M$ is the least upper bound of $M$ and $inf\ M$ is the greatest lower bound of $M$. The worst-case cost of an initial configuration $C$ is the cost of the most expensive complete trace starting from $C$ and the best-case cost is the less expensive complete trace.

**Definition 1 (worst- and best-case cost).** *Let $\mathcal{S}$ be a TS. Its worst-case cost function $wc_{\mathcal{S}} : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is $wc_{\mathcal{S}}(C) = sup\ \{len(t) \mid t \in Traces_{\mathcal{S}}(C)\}$ and its best-case cost function $bc_{\mathcal{S}} : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is $bc_{\mathcal{S}}(C) = inf\ \{len(t) \mid t \in Traces_{\mathcal{S}}(C)\}$.*

Clearly, $wc_{\mathcal{S}}$ and $bc_{\mathcal{S}}$ are not computable. Our goal in this paper is to automatically find a lower-bound function $\rho : \mathbb{Z}^n \to \mathbb{R}_{\geq 0}$ such that for any initial configuration $C = (\ell_0, \sigma)$ we have $wc_{\mathcal{S}}(C) \geq \rho(\sigma(\bar{x}))$, i.e., it is an LB$^w$. An LB$^b$ would be a function $\rho : \mathbb{Z}^n \to \mathbb{R}_{\geq 0}$ that ensures that $bc_{\mathcal{S}}(C) \geq \rho(\bar{x})$ for any initial configuration $C = (\ell_0, \sigma)$. In what follows, for a function $\rho(\bar{x})$, we let $\|\rho(\bar{x})\| = \lceil \max(0, \rho(x)) \rceil$ to map all negative valuations of $\rho$ to zero.

*Example 2.* Consider the TS $\mathcal{S} = \langle \{x\}, \{u\}, \{\ell_0, \ell_1, \ell_e\}, \mathcal{T}, true \rangle$ with transitions:

$$\begin{aligned} \mathcal{T} \equiv \{\ & \tau_1 = (\ell_0, \ell_1, x \geq 0), \\ & \tau_2 = (\ell_1, \ell_1, x > 0 \wedge x' = x - u \wedge u \geq 1 \wedge u \leq 2), \\ & \tau_3 = (\ell_1, \ell_e, x \leq 0 \wedge x' = x)\ \} \end{aligned}$$

$\mathcal{S}$ contains a loop at $\ell_1$ where variable $x$ is non-deterministically decreased by 1 or 2. From any initial configuration $C_0 = (\ell_0, \sigma_0)$, the longest possible complete trace decreases $x$ by 1 in every iteration with $\tau_2$, therefore $wc_{\mathcal{S}}(C_0) = \|\sigma_0(x)\| + 2$ because of the $\|\sigma_0(x)\|$ iterations in $\ell_1$ plus the cost of $\tau_1$ and $\tau_3$. The most precise lower bound for $wc_{\mathcal{S}}$ is $\rho(x) = \|x\| + 2$, although $\rho(x) = \|x\|$ or $\rho(x) = \|x - 2\|$ are also valid lower bounds. The shortest complete trace from $C_0$ decreases $x$ by 2 in every iteration, so $bc_{\mathcal{S}}(C_0) = \|\frac{\sigma_0(x)}{2}\| + 2$. There are several valid lower bounds for $bc_{\mathcal{S}}(C_0)$ like $\rho(x) = \|\frac{x}{2}\| + 2$, $\rho(x) = \|\frac{x}{2}\|$, or $\rho(x) = 2$.

## 3 Local Lower-Bound Functions

*Focus on Local Bounds.* Existing techniques and tools for cost analysis (e.g., [1, 12]) work by inferring *local* (iteration) bounds for those parts of the TS that correspond to loops, and then combining these bounds by propagating them "backwards" to the entry point in order to obtain a *global* bound. For example, suppose that our program consists of the following two loops:

```
assert(x>0 && z>0);
while (z > 0) { x=x+z; z--; }
while (x > 0) x--;
```

where the second loop makes $x$ iterations (when considering the value of $x$ just before executing the loop), and the first loop makes $z$ iterations and increments $x$ by $z$ in each iteration. We are interested in inferring a global function that describes the total number of iterations of both loops, in terms of the input values $x_0$ and $z_0$. While both loops have linear complexity locally, i.e., iteration bounds $z$ and $x$, the second one has quadratic complexity w.r.t the initial values. This can be inferred automatically from the local bounds $z$ and $x$ by inferring how the value of $x$ changes in the first loop, and then rewriting $x$ in terms of the initial values to $e = x_0 + \frac{z_0 \cdot (z_0-1)}{2}$ (e.g., by solving corresponding recurrence relations). Now the global cost would be $e$ plus the cost of the first loop $z_0$. Rewriting the loop bound $x$ as above is done by propagating it backwards to the entry point, and there are several techniques in the literature for this purpose that can be directly adopted in our setting to produce global bounds. These techniques can infer global bounds for nested-loops as well, given the iteration bounds of each loop. Thus, we focus on inferring local lower-bounds on the number of iterations that non-nested loops (more precisely, parts of the TS that correspond to loops) can make, and assume that they can be rewritten to global bounds by adopting the existing techniques of [1,12] (our implementation indeed could be used as a black-box which provides local lower-bounds to these tools). Namely, we aim at inferring, for each non-nested loop, a function $\|\rho(\bar{x})\| = \lceil \max(0, \rho(x)) \rceil$ that is a (local) $\mathrm{LB}^w$ on its number of iterations, i.e., whenever the loop is reached with values $\bar{v}$ for the variables $\bar{x}$, it is possible to make at least $\|\rho(\bar{v})\|$ iterations.

*Loops and TSs.* For ease of presentation, we first consider a special case of TSs in which all locations, except the initial and exit ones define loops, and Sect. 3.6 explains how the techniques can be used for the general case. In particular, we consider that each non-trivial SCC consists of a single location $\ell$ and at least one transition, and we call it *loop* $\ell$. Transitions from $\ell$ to $\ell$ are called *loop transitions* and their guards are called *loop guards*, and transitions from $\ell$ to $\ell' \neq \ell$ are called *exit transitions*. The number of iterations of a loop $\ell$ in a trace $t$ is defined as the number of transitions from $\ell$ to $\ell$, which we refer to as the cost of loop $\ell$ as well (since we are assuming that the cost of transitions is always 1, see Sect. 2.2). The notions of best-case and worst-case cost in Definition 1 naturally extend to the cost of a loop $\ell$, i.e., we can ask what is the best-case and worst-case number of iterations of a given loop.

*Overview of the Section.* The overall idea of our approach is to *specialize* each loop $\ell$, by restricting the initial values and/or adding constraints to its transitions, such that it becomes possible to obtain a metering function for the specialized loop. A function that is a LB$^b$ of the specialized loop is by definition a LB$^w$ of loop $\ell$, as it does not necessarily hold for all execution traces but rather for the class of restricted ones. Technically, inferring a LB$^b$ of a (specialized) loop is done by inferring a metering function $\rho$ [13], such that whenever the (specialized) loop is reached with a state $\sigma$, it is guaranteed to make at least $\|\rho(\sigma(\bar{x}))\|$ iterations. Besides, specialization is done in such away that the TS obtained by putting all specialized loops together is non-blocking, i.e., there is an execution that is either non-terminating or reaches the exit location, and thus the cost of this execution is, roughly, the sum of the costs of all (specialized) loops that are traversed. The rest of this section is organized as follows. In Sect. 3.1 we generalize the basic definition of metering function for simple loops from [12] to general types of loops and explore its limitations. Then, in the following 3 sections, we explain how to overcome these limitations by means of the following specializations: using quasi-invariants to narrow the set of input values (Sect. 3.2); narrowing loop guards to make loop transitions mutually exclusive and force some execution order between them (Sect. 3.3); and narrowing the space of non-deterministic choices to force longer executions (Sect. 3.4). Sect. 3.5 states the conditions, to be satisfied when specializing loops, in order to guarantee that the TS obtained by putting all specialized loops together is non-blocking.

## 3.1 Metering Functions

*Metering functions* were introduced by [13], as a tool for inferring a lower-bound on the number of iterations that a given loop can make. The definition is analogue to that of (linear) ranking function which is often used to infer upper-bounds on the number of iterations. The definition as given in [13] considers a loop with a single transition, and assumes that the exit condition is the negation of its guard. We start by generalizing it to our notion of loop.

**Definition 2 (Metering function).** *We say that a function $\rho_\ell$ is a metering function for a loop $\ell \in \mathcal{L}$, if the following conditions are satisfied*

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{3}$$

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{R} \rightarrow \rho_\ell(\bar{x}) \leq 0 \qquad \textbf{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \tag{4}$$

*Intuitively, Condition* (3) *requires $\rho_\ell$ to decrease at most by* 1 *in each iteration, and Condition* (4) *requires $\rho_\ell$ to be non-positive when leaving the loop.*

Assuming $(\ell, \sigma)$ is a reachable configuration in $\mathcal{S}$, it is easy to see that loop $\ell$ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations when starting from $(\ell, \sigma)$. We require $(\ell, \sigma)$ to be reachable in $\mathcal{S}$ since we are interested only in non-blocking executions. Typically, we are interested in linear metering functions, i.e., of the form $\rho_\ell(\bar{x}) = \bar{a} \cdot \bar{x} + a_0$, since they are easier to infer and cover most loops in practice. Non-linear lower-bound functions will be obtained when rewriting these local linear lower-bounds in terms of the initial input at location $\ell_0$ (see beginning of Sect. 3) and by composing nested loops (see Sect. 3.6).

*Example 3 (Metering function).* Consider the following loop on location $\ell_1$ that decreases $x$ ($\tau_1$) until it takes non-positive values and exits to $\ell_2$ ($\tau_2$):

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge x' = x - 1) \qquad \tau_2 = (\ell_1, \ell_2, x < 0 \wedge x' = x)$$

The function $\rho_{\ell_1}(x) = x + 1$ is a valid metering function because it decreases by exactly 1 in $\tau_1$ and becomes non-positive when $\tau_2$ is applicable ($x < 0 \rightarrow x+1 \leq 0$, Condition (3) of Definition 2). The function $\rho'_{\ell_1}(x) = \frac{x}{2}$ is also metering because its value decreases by less than 1 when applying $\tau_1$ ($\frac{x}{2} - \frac{x-1}{2} = \frac{1}{2} \leq 1$) and becomes non-positive in $\tau_2$. Even a function as $\rho''_{\ell_1}(x) = 0$ is trivially metering, as it satisfies (3) and (4). Although all of them are valid metering functions, $\rho_{\ell_1}(x)$ is preferable as it is more accurate (i.e., larger) and thus captures more precisely the number of iterations of the loop. Note that functions like $\rho^*_{\ell_1}(x) = 2x$ or $\rho^{**}_{\ell_1}(x) = x + 5$ are not metering because they do not verify (3) (because $2x - 2(x-1) = 2 \not\leq 1$ for $\rho^*_{\ell_1}$) or (4) (because $x < 0 \not\rightarrow x + 5 \leq 0$ for $\rho^{**}_{\ell_1}$).

## 3.2 Narrowing the Set of Input Values Using Quasi-Invariants

Metering functions typically exist for loops with simple loop guards. However, when guards involve more than one inequality they usually do not exist in a simple (linear) form. This is because such loops often include several exit transitions with unrelated conditions, where each one corresponds to the negation of an inequality of the guard. It is unlikely then that a non-trivial (linear) function satisfies (4) for all exit transitions. This is illustrated in the next example.

*Example 4.* Consider the following loop that iterates on $\ell_1$ if $x \geq 0 \wedge y > 0$, and exits when $x < 0$ or $y \leq 0$:

$$\begin{aligned}\tau_1 &= (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y)\\ \tau_2 &= (\ell_1, \ell_2, x < 0 \wedge x' = x \wedge y' = y)\\ \tau_3 &= (\ell_1, \ell_2, y \leq 0 \wedge x' = x \wedge y' = y)\end{aligned}$$

Intuitively, this loop executes $x + 1$ transitions, but $\rho_{\ell_1}(x, y) = x + 1$ is not a valid metering function because it does not satisfy (4) for $\tau_3$: $y \leq 0 \not\rightarrow x+1 \leq 0$. Moreover, no other function depending on $x$ (e.g., $\frac{x}{2}$, $x - 2$, etc.) will be a valid metering function, as it will be impossible to prove (4) for $\tau_3$ only from the information $y \leq 0$ on its guard. The only valid metering function for this loop will be the trivial one $\rho_{\ell_1}(x, y) = c$ with $c \leq 0$, which does not provide any information about the number of iterations of the loop.

Our proposal to overcome the imprecision discussed above is to consider only a subset of the input values s.t. conditions (3,4) hold in the context of the corresponding reachable states. For example, the reachable states might exclude some of the exit transitions, i.e., it is guaranteed that they are never used, and then (4) is not required to hold for them. A metering function in this context is a LB$^b$ of the loop when starting from that specific input, and thus it is a LB$^w$ (i.e., not necessarily best-case) of the loop when the input values are not restricted.

Technically, our analysis materializes the above idea by relying on quasi-invariants [17]. A quasi-invariant for a loop $\ell$ is a formula $\mathcal{Q}_\ell$ over $\bar{x}$ such that

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{5}$$

$$\exists \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \tag{6}$$

Intuitively, $\mathcal{Q}_\ell$ is similar to an inductive invariant but without requiring it to hold on the initial states, i.e., once $\mathcal{Q}_\ell$ holds it will hold during all subsequent visits to $\ell$. This also means that for executions that start in states within $\mathcal{Q}_\ell$, it is guaranteed that $\mathcal{Q}_\ell$ is an over-approximation of the reachable states. Condition (6) is used to avoid quasi-invariants that are *false*. Given a quasi-invariant $\mathcal{Q}_\ell$ for $\ell$, we say that $\rho_\ell$ is a metering function for $\ell$ if the following holds

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{7}$$

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) \leq 0 \qquad \textbf{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \tag{8}$$

Intuitively, these conditions state that (3,4) hold in the context of the states induced by $\mathcal{Q}_\ell$. Assuming that $(\ell, \sigma)$ is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, loop $\ell$ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in $(\ell, \sigma)$.

*Example 5.* Recall that the loop in Example 4 only admitted trivial metering functions because of the exit transition $\tau_3$. It is easy to see that $\mathcal{Q}_{\ell_1} \equiv x < y$ verifies (5,6), because $y$ is not modified in $\tau_1$ and $x$ decreases, and thus it is a quasi-invariant. In the context of $\mathcal{Q}_{\ell_1}$, function $\rho_{\ell_1}(x, y) = x + 1$ is metering because when taking $\tau_3$ the value of $x$ is guaranteed to be negative, i.e., $\tau_3$ satisfies (8) because $x < y \wedge y \leq 0 \rightarrow x + 1 \leq 0$. Notice that $\rho_{\ell_1}(x, y) = x + 1$ will still be a valid metering function considering other quasi-invariants of the form $\mathcal{Q}'_{\ell_1} \equiv y > c$ with $c \geq 0$, as they would completely disable transition $\tau_3$.

## 3.3 Narrowing Guards

The loops that we have considered so far consist of a single loop transition, what makes easier to find a metering function. This is because there is only one way to modify the program variables (with some degree of non-determinism induced by the non-deterministic variables). However, when we allow several loop transitions, we can have loops for which a non-trivial metering function does not exist even when narrowing the set of input values.

*Example 6.* Consider the extension of the loop in Example 4 with a new transition $\tau_4$ that decrements $y$ (it corresponds to the example in Sect. 1):

$$\begin{aligned}
&\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x - 1 \wedge y' = y)\\
&\tau_4 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x \wedge y' = y - 1)\\
&\tau_2 = (\ell_1, \ell_2, x < 0 \wedge x' = x \wedge y' = y)\\
&\tau_3 = (\ell_1, \ell_2, y \leq 0 \wedge x' = x \wedge y' = y)
\end{aligned}$$

The most precise LB$^w$ of this loop is $\|\rho_{\ell_1}(x, y)\|$ where $\rho_{\ell_1}(x, y) = x + y$. As mentioned, this corresponds, e.g., to an execution that uses $\tau_1$ until $x = 0$, i.e., $x$ times, and then $\tau_4$ until $y = 0$, i.e., $y$ times. It is easy to see that if we start from a state that satisfies $x \geq 0 \wedge x \leq y$, then it will be satisfied during the particular execution that we just described. Moreover, assuming that $\mathcal{Q}_{\ell_1} \equiv x \geq 0 \wedge x \leq y$ is a quasi-invariant, it is easy to show that together with $\rho_{\ell_1}$ we can verify (7,8), and thus $\rho_{\ell_1}$ will be a metering function. However, unfortunately, $\mathcal{Q}_{\ell_1}$ is not a quasi-invariant since the above loop can make executions other than the one described above (e.g., decreasing $y$ to 1 first and then $x$ to 0).

Our idea to overcome this imprecision is to narrow the set of states for which loop transitions are enabled, i.e., strengthening loop guards by additional inequalities. This, in principle, reduces the number of possible executions, and thus it is more likely to find a metering function (or a better quasi-invariant), because now they have to be valid for fewer executions. For example, this might force an execution order between the different paths, or even disable some transitions by narrowing their guard to *false*. Again, a metering function for the specialized loop is not a valid LB$^b$ of the original loop, but rather its a valid LB$^w$ that is what we are interested in. Next, we state the requirements that such narrowing should satisfy. The choice of a narrowing that leads to longer executions is discussed in Sect. 4.

A *guard narrowing* for a loop transition $\tau \in \mathcal{T}$ is a formula $\mathcal{G}_\tau(\bar{x})$, over variables $\bar{x}$. A specialization of a loop is obtained simply by adding these formulas to the corresponding transitions. Conditions (5)-(8) can be specialized to hold only for executions that use the specialized loop as follows. Suppose that for a loop $\ell \in \mathcal{L}$ we are given a narrowing $\mathcal{G}_\tau$ for each loop transition $\tau$, then $\mathcal{Q}_\ell$ and $\rho_\ell$ are quasi-invariant and metering function resp. for the corresponding specialized loop if the following conditions hold

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{9}$$

$$\exists \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \tag{10}$$

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \leq 1 \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \tag{11}$$

$$\forall \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \rightarrow \rho_\ell(\bar{x}) \leq 0 \qquad \textbf{for each } (\ell, \ell', \mathcal{R}) \in \mathcal{T} \tag{12}$$

Conditions (9,10) guarantee that $\mathcal{Q}_\ell$ is a non-empty quasi-invariant for the specialized loop, and conditions (11,12) guarantee that $\rho_\ell$ is a metering function for the specialized loop in the context of $\mathcal{Q}_\ell$. However, in this case, function $\rho_\ell$ induces a lower-bound on the number of iterations only if the specialized loop is non-blocking for states in $\mathcal{Q}_\ell$. This is illustrated in the following example.

*Example 7.* Consider the loop from Example 3 where we have specialized the guard of $\tau_1$ by adding $x \geq 5$:

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge x \geq 5 \wedge x' = x - 1) \qquad \tau_2 = (\ell_1, \ell_2, x < 0 \wedge x' = x)$$

With this specialized guard and considering $\mathcal{Q}_{\ell_1} \equiv true$, the metering function $\rho_{\ell_1}(x) = x + 1$ still satisfies (11,12), and $\mathcal{Q}_{\ell_1}$ trivially satisfies (9,10). However, $\rho_{\ell_1}$ is not a valid measure of the number of transitions executed because the loop gets blocked whenever $x$ takes values $0 \leq x \leq 5$, and thus it will never execute $x + 1$ transitions.

To guarantee that the specialized loop is non-blocking for states in $\mathcal{Q}_\ell$, it is enough to require the following condition to hold

$$\forall \bar{x}.\ \mathcal{Q}_\ell(\bar{x}) \rightarrow \bigvee_{\tau=(\ell,\ell,\mathcal{R})\in\mathcal{T}} (\mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})) \bigvee_{\tau=(\ell,\ell',\mathcal{R})\in\mathcal{T}} \mathcal{R}(\bar{x}) \tag{13}$$

Intuitively, it states that from any state in $\mathcal{Q}_\ell$ we can make progress, either by making a loop iteration or exiting the loop. Assuming that $(\ell,\sigma)$ is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, the specialized loop $\ell$ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in $(\ell,\sigma)$. This also means that the original loop *can* make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in $(\ell,\sigma)$.

*Example 8.* In Example 6, we have seen that if $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ was a quasi-invariant, then function $\rho_{\ell_1}(x,y) = x+y$ becomes metering. We can make $\mathcal{Q}_{\ell_1}$ a quasi-invariant by specializing the guards of the loop in transitions $\tau_1$ and $\tau_4$ to force the following execution with $x+y$ iterations: first use $\tau_1$ until $x=0$ ($x$ iterations) and then use $\tau_4$ until $y=0$ ($y$ iterations). This behavior can be forced by taking $\mathcal{G}_{\tau_1} \equiv x > 0$ and $\mathcal{G}_{\tau_4} \equiv x \leq 0$. With $\mathcal{G}_{\tau_1}$ we assure that $x$ stops decreasing when $x=0$, and with $\mathcal{G}_{\tau_4}$ we assure that $\tau_4$ is used only when $x=0$. Now, $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ and $\rho_{\ell_1}(x,y)=x+y$ are valid quasi-invariant and metering, resp. Function $\rho_{\ell_1}$ decreases by exactly 1 in $\tau_1$ and $\tau_4$, is trivially non-positive in $\tau_2$ because that transition is indeed disabled ($x \geq 0$ from $\mathcal{Q}_{\ell_1}$ and $x<0$ from the guard) and is non-positive in $\tau_3$ ($x \leq y \wedge y \leq 0 \rightarrow x+y \leq 0$). Regarding $\mathcal{Q}_{\ell_1}$, it verifies (9,10), and more importantly, the loop in $\ell_1$ is non-blocking w.r.t $\mathcal{Q}_{\ell_1}$, $\mathcal{G}_{\tau_1}$, and $\mathcal{G}_{\tau_4}$, i.e., Condition (13) holds.

## 3.4 Narrowing Non-deterministic Choices

Loop transitions that involve non-deterministic variables, might give rise to executions of different lengths when starting from the same input values. Since we are interested in $\text{LB}^w$, we are clearly searching for longer executions. However, since our approach is based on inferring $\text{LB}^b$, we have to take all executions into account which might result in less precise, or even trivial, $\text{LB}^w$.

*Example 9.* Consider a modification of the loop in Example 6 in which the variable $x$ in $\tau_1$ is decreased by a non-deterministic positive quantity $u$:

$$\tau_1 = (\ell_1, \ell_1, x \geq 0 \wedge y > 0 \wedge x' = x-u \wedge u \geq 1 \wedge y' = y)$$

The effect of this non-deterministic variable $u$ is that $\tau_1$ can be applied $x$ times if we always take $u=1$, $\lceil \frac{x}{2} \rceil$ times if we always take $u=2$ or even only once if we take $u > x$. As a consequence, $\rho_{\ell_1}(x,y)=x+y$ is no longer a valid metering function because $x$ can decrease by more than 1 in $\tau_1$. Moreover, $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$ is not a quasi-invariant anymore since $x' = x-u \wedge u \geq 1$ does not entail $x' \geq 0$. In fact, no metering function involving $x$ will be valid in $\tau_1$ because $x$ can decrease by any positive amount.

To handle this complex situation, we propose narrowing the space of non-deterministic choices, and thus metering functions should be valid *wrt.* fewer executions and more likely be found and be more precise. Next we state the requirements that such narrowing should satisfy. The choice of a narrowing that leads to longer executions is discussed in Sect. 4.

A *non-deterministic variables narrowing* for a loop transition $\tau \in \mathcal{T}$ is a formula $\mathcal{U}_\tau(\bar{x}, \bar{u})$, over variables $\bar{x}$ and $\bar{u}$, that is added to $\tau$ to restrict the choices for variables $\bar{u}$. A specialized loop is now obtained by adding both $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ to the corresponding transitions. Suppose that for loop $\ell \in \mathcal{L}$, in addition to $\mathcal{G}_\tau$, we are also given $\mathcal{U}_\tau$ for each of its loop transitions $\tau$. For $\mathcal{Q}_\ell$ and $\rho_\ell$ to be quasi-invariant and metering function for the specialized loop $\ell$, we require conditions (9)-(13) to hold but after adding $\mathcal{U}_\tau$ to the left-hand side of the implications in (9) and (11). Besides, unlike narrowing of guards, narrowing of non-deterministic choices might make a transition invalid, i.e., not satisfying Condition (1), and thus $\|\rho_\ell(\bar{x})\|$ cannot be used as a lower-bound on the number of iterations. To guarantee that specialized transitions are valid we require, in addition, the following condition to hold

$$\forall \bar{x} \exists \bar{u}.\ \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \rightarrow \mathcal{R}(\bar{x}, \bar{u}) \wedge \mathcal{U}_\tau(\bar{x}, \bar{u}) \qquad \textbf{for each } (\ell, \ell, \mathcal{R}) \in \mathcal{T} \qquad (14)$$

This condition is basically (1) taking into account the inequalities introduced by the corresponding narrowings. Assuming that $(\ell, \sigma)$ is reachable in $\mathcal{S}$ and that $\sigma \models \mathcal{Q}_\ell$, the specialized loop $\ell$ will make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in $(\ell, \sigma)$, which also means, as before, that the original loop *can* make at least $\|\rho_\ell(\sigma(\bar{x}))\|$ iterations in any execution that starts in $(\ell, \sigma)$.

*Example 10.* To solve the problems shown in Example 9 we need to narrow the non-deterministic variable $u$ to take bounded values that reflect the worst-case execution of the loop. Concretely, we need to take $\mathcal{U}_{\tau_1} \equiv u \leq 1$, which combined with $u \geq 1$ entails $u = 1$ so $x$ decreases by exactly 1 in $\tau_1$. Considering the narrowing $\mathcal{U}_{\tau_1}$, the resulting loop is equivalent to the one presented in Example 8 so we could obtain the precise metering function $\rho_{\ell_1}(x, y) = x + y$ with the quasi-invariant $\mathcal{Q}_{\ell_1} \equiv x \leq y \wedge x \geq 0$. Note that (14) holds for $\tau_1$ because $u = 1$ makes the consequent true for every value of $x$ and $y$: $\forall \bar{x} \exists \bar{u}.\ (x \leq y \wedge x \geq 0) \wedge (x \geq 0 \wedge y > 0) \wedge x > 0 \rightarrow u \geq 1 \wedge u \leq 1$

## 3.5 Ensuring the Feasibility of the Specialized Loops

In order to enable the propagation of the local lower-bounds back to the input location (as we have discussed at the beginning of Sect. 3), we have to ensure that there is actually an execution that starts in $\ell_0$ and passes through the specialized loop. In other words, we have to guarantee that when putting all specialized loops together, they still form a non-blocking TS for some set of input values. We achieve this by requiring that the quasi-invariants of the preceding loops ensure that the considered quasi-invariant for this loop also holds on initialization (i.e., it is an invariant for the considered context). Technically, we require, in addition to (9)-(14), the following conditions to hold for each loop $\ell$:

$$\forall \bar{x}, \bar{u}, \bar{x}'.\ \mathcal{Q}_{\ell'}(\bar{x}) \wedge \mathcal{R} \rightarrow \mathcal{Q}_\ell(\bar{x}') \qquad \textbf{for each } (\ell', \ell, \mathcal{R}) \in \mathcal{T} \qquad (15)$$

$$\forall \bar{x}.\ \mathcal{Q}_{\ell_0} \rightarrow \Theta \qquad (16)$$

Condition (15) means that transitions entering loop $\ell$, strengthened with the quasi-invariant of the preceding location $\ell'$, must lead to states within the quasi-invariant $\mathcal{Q}_\ell$. Condition (16) guarantees that $\mathcal{Q}_{\ell_0}$ defines valid input values, i.e., within the initial condition $\Theta$.

**Theorem 1 (soundness).** *Given $\mathcal{Q}_\ell$ for each non-exit location $\ell \in \mathcal{L}$, narrowings $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ for each loop transition $\tau \in \mathcal{T}$, and function $\rho_\ell$ for each loop location $\ell$, such that* (9)-(16) *are satisfied, it holds:*

1. *The TS $\mathcal{S}'$ obtained from $\mathcal{S}$ by adding $\mathcal{G}_\tau$ and $\mathcal{U}_\tau$ to the corresponding transitions, and changing the initial condition to $\mathcal{Q}_{\ell_0}$, is non-blocking.*
2. *For any complete trace $t$ of $\mathcal{S}'$, if $C = (\ell, \sigma)$ is a configuration in $t$, then $t$ includes at least $\|\rho_\ell(\sigma(\bar{x}))\|$ visits to $\ell$ after $C$ (i.e., $\|\rho_\ell(\bar{x})\|$ is a lower-bound function on the number of iterations of the loop defined by location $\ell$).*

The proof of this soundness result is straightforward: it follows as a sequence of facts using the definitions of the conditions (9)-(16) given in this section.

We note that when there is an unbounded overlap between the guards of the loop transitions and the guards of exit transitions, it is likely that a non-trivial metering function does not exist because it must be non-positive on the overlapping states. To overcome this limitation, instead of using the exit transitions in (12), we can use ones that correspond to the negation of the guards of loop transitions, and thus it is ensured that they do not overlap. However, we should require (13) to hold for the original exit transitions as well in order to ensure that the non-blocking property holds. Another way to overcome this limitation is to simply strengthen the exit transitions by the negation of the guards.

As a final comment, we note that it is not needed to assume that the TS $\mathcal{S}$ that we start with is non-blocking (even though we have done so in Sect. 2.1 for clarity). This is because our formalization above finds a subset of $\mathcal{S}$ ($\mathcal{S}'$ in Theorem 1) that is non-blocking, which is enough to ensure the feasibility of the local lower-bounds. This is useful not only for enlarging the set of TSs that we accept as input, but also allows us to start the analysis from any subset of $\mathcal{S}$ that includes a path from $\ell_0$ to the exit location. For example, it can be used to remove trivial execution paths from $\mathcal{S}$, or concentrate on ones that include more sequences of loops (since we are interested in LB$^w$).

## 3.6 Handling General TSs

So far we have considered a special case of TSs in which all locations, except the entry and exit ones, are multi-path loops. Next we explain how to handle the general case. It is easy to see that we can allow locations that correspond to trivial SCCs. These correspond to paths that connect loops and might include branching as well. For such locations, there is no need to infer metering functions or apply any specialization, we only need to assign them quasi-invariants that satisfy (15) to guarantee that the overall specialized TS is non-blocking.

The more elaborated case is when the TS includes non-trivial SCCs that do not form a multi-path loop. In such case, if a SCC has a single cut-point, we can unfold its edges and transform it into a multi-path following the techniques of [1]. It is important to note that when merging two transitions, the cost of the new one is the sum of their costs. In this case the number of iterations is still a lower-bound on the cost of the loop, however, we might get a better one by multiplying it by the minimal cost of its transitions.

If a SCC cannot be transformed into a multi-path loop by unfolding its transitions, then it might correspond to a nested loop, and, in such case, we can recover the nesting structure and consider them as separated TSs that are "called" from the outer one using loop extraction techniques [25]. Each inner-loop is then analyzed separately, and replaced (in the original TS, where is "called") by a single edge with its lower-bound as cost for that edge, and then the outer is analyzed taking that cost into account. Besides, to guarantee that the specialized program corresponds to a valid execution, we require the quasi-invariant of the inner loop to hold in the context of the quasi-invariant of the outer loop. This approach is rather standard in cost analysis of structured programs [1,3,12].

Another issue is how to compose the (local) lower-bounds of the specialized loops into a global-lower bound. For this, we can rely on the techniques [1,3] that rewrite the local lower-bounds in terms of the input values by relying on invariant generation and recurrence relations solving.

## 4 Inference Using Max-SMT

This section presents how metering functions and narrowings can be inferred automatically using Max-SMT, namely how to automatically infer all $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, $\mathcal{Q}_\ell$, and $\rho_\ell$ such that (9)-(16) are satisfied. We do it in a modular way, i.e., we seek $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, $\mathcal{Q}_\ell$, and $\rho_\ell$ for one loop at a time following a (reversed) topological order of the SCCs, as we describe next. Recall that (16) is required only for loops connected directly to $\ell_0$, and w.l.o.g. we assume there is only one such loop.

### 4.1 A Template-Based Verification Approach

We first show how the template-based approach of [6,17] can be used to find $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$ by representing them as template constraint systems, i.e., each is a conjunction of linear constraints where coefficients and constants are unknowns. Also, $\rho_\ell$ is represented as a linear template function $\bar{a} \cdot \bar{x} + a_0$ where $(a_0, \bar{a})$ are unknowns. Then, the problem is to find concrete values for the unknowns such that all formulas generated by (9)-(16) are satisfied:

- Each $\forall$-formula generated by (9)-(16), except those of (14) that we handle below, can be viewed as an $\exists\forall$ problem where the $\exists$ is over the unknowns of the templates and the $\forall$ is over (some of) the program variables. It is well-known that solving such an $\exists\forall$ problem, i.e., finding values for the unknowns, can be done by translating it into a corresponding $\exists$ problem over the existentially quantified variables (i.e., the unknowns) using Farkas' lemma [20], which can then be solved using an off-the-shelf SMT solver.

- To handle (14) we follow [17], and eliminate $\exists\bar{u}$ using the skolemization $u_i = \bar{a} \cdot \bar{x} + a_0$ where $(a_0, \bar{a})$ are fresh unknowns (different for each $u_i$). This allows handling it using Farkas' lemma as well. However, in addition, when solving the corresponding $\exists$ problem we require all $(a_0, \bar{a})$ to be integer. This is because the domain of program variables is the integers, and picking integer values for all $(a_0, \bar{a})$ guarantees that the values of any $x'_i$ that depends on $\bar{u}$ will be integer as well[1].

The size of templates for $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$, i.e., the number of inequalities, is crucial for precision and performance. The larger the size is, the more likely that we get a solution if one exists, but also the worse the performance is (as the corresponding SMT problem will include more constraints and variables). In practice, one typically starts with templates of size 1, and iteratively increases it by 1 when failing to find values for the unknowns, until a solution is found or the bound on the size is reached.

Alternatively, we can use the approach of [17] to construct $\mathcal{G}_\tau$, $\mathcal{U}_\tau$, and $\mathcal{Q}_\ell$ incrementally. This starts with templates of size 1, but instead of requiring all (9)-(16) to hold, the conditions generated by (12) are marked as soft constraints (i.e., we accept solutions in which they do not hold) and use Max-SMT to get a solution that satisfies as many of such soft conditions as possible. If all are satisfied, we are done, if not, we use the current solution to instantiate the templates, and then add another template inequality to each of them and repeat the process again. This means that at any given moment, each template will include at most one inequality with unknowns. Finally, to guarantee progress from one iteration to another, soft conditions that hold at some iteration are required to hold at the next one, i.e., they become hard.

The use of (12) as soft constraint is based on the observation [12] that when seeking a metering function, the problematic part is often to guarantee that it is negative on exit transitions, which is normally achieved by adding quasi-invariants that are incrementally inferred. By requiring (12) to be soft we handle more exit transitions as the quasi-invariant gets stronger until all are covered.

## 4.2 Better Quality Solutions

The precision can also be affected by the quality of the solution picked by the SMT solver for the corresponding $\exists$ problem. Since there might be many metering functions that satisfy (9)-(16), we are interested in narrowing the search space of the SMT solver in order to find more accurate ones, i.e., lead to longer executions. Next we present some techniques for this purpose.

*Enabling More Loop Transitions.* We are interested in guard narrowings that keep as many loop transitions as possible, since such narrowings are more likely

[1] Because we assumed that constraints involving primed variables are of the form $x'_i = \bar{a} \cdot \bar{x} + \bar{b} \cdot \bar{u} + c$.

to generate longer executions. This can be done by requiring the following to hold

$$\exists \bar{x}. \bigvee_{\tau=(\ell,\ell,\mathcal{R})\in\mathcal{T}} (\mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})) \tag{17}$$

We also use Max-SMT to require a solution that satisfies as many disjuncts as possible and thus eliminating less loop transitions (if $\mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{R}(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x})$ is *false* for a transition $\tau$, then it is actually disabled). Note that this condition can be used instead of (10) that requires the quasi-invariant to be non-empty.

*Larger Metering Functions.* We are interested in metering functions that lead to longer executions. One way to achieve this is to require metering functions to be ranking as well, i.e., in addition to (11) we require the following to hold

$$\forall \bar{x}, \bar{u}, \bar{x}'. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{U}_\tau(\bar{x}, \bar{u}) \wedge \mathcal{R} \rightarrow \rho_\ell(\bar{x}) - \rho_\ell(\bar{x}') \geq 1 \quad \textbf{for each } (\ell,\ell,\mathcal{R}) \in \mathcal{T} \tag{18}$$

$$\forall \bar{x}, \bar{u}. \mathcal{Q}_\ell(\bar{x}) \wedge \mathcal{G}_\tau(\bar{x}) \wedge \mathcal{R}(\bar{x}) \rightarrow \rho_\ell(\bar{x}) \geq 0 \quad \textbf{for each } (\ell,\ell,\mathcal{R}) \in \mathcal{T} \tag{19}$$

These new conditions are added as soft constraints, and we use Max-SMT to ask for a solution that satisfies as many conditions as possible.

*Unbounded Metric Functions.* We are interested in metering functions that do not have an upper bound, since otherwise they will lead to constant lower-bound functions. For example, for a loop with a transition $x \geq 0 \wedge x' = x - 1$, we want to avoid quasi-invariants like $x \leq 5$ which would make the metering function $x$ bounded by 5. For this, we rely on the following lemma.

**Lemma 1.** *A function $\rho(\bar{x}) = \bar{a} \cdot \bar{x} + a_0$ is unbounded over a polyhedron $\mathcal{P}$, iff $\bar{a} \cdot \bar{y}$ is positive on at least one ray $\bar{y}$ of the* recession cone *of $\mathcal{P}$.*

It is known that for a polyhedron $\mathcal{P}$ given in constraints representation, its recession cone $\texttt{cone}(\mathcal{P})$ is the set specified by the constraints of $\mathcal{P}$ after removing all free constants. Now we can use the above lemma to require that the metering function $\rho_\ell(\bar{x}) = \bar{a} \cdot \bar{x} + \bar{a}_0$ is unbounded in the quasi-invariant $\mathcal{Q}_\ell$ by requiring the following condition to hold

$$\exists \bar{x}.\ \texttt{cone}(\mathcal{Q}_\ell) \wedge \bar{a} \cdot \bar{x} > 0 \tag{20}$$

where $\texttt{cone}(\mathcal{Q}_\ell)$ is obtained from the template of $\mathcal{Q}_\ell$ by removing all (unknowns corresponding to) free constants, i.e., it is the *recession cone* of $\mathcal{Q}_\ell$.

Note that all encodings discussed in this section generate non-linear SMT problems, because they either correspond to $\exists\forall$ problems that include templates on the left-hand side of implications, or to $\exists$ problems over templates that include both program variables and unknowns.

Finally, it is important to note that the optimizations described provide theoretical guarantees to get better lower bounds: the one that adds (18,19) leads to a bound that corresponds exactly to the worst-case execution (of the specialized program), and the one that uses (20) is essential to avoid constant bounds.

## 5 Implementation and Experimental Evaluation

We have implemented a *LO*wer-*B*ound synthesiz*ER*, named LOBER, that can be used from an online web interface at http://costa.fdi.ucm.es/lober. LOBER is built as a pipeline with the following processes: (1) it first reads a KoAT file [5] and generates a corresponding set of multi-path loops, by extracting parts of the TS that correspond to loops [25], applying unfolding, and inferring loop summaries to be used in the calling context of nested loops, as explained in Sect. 3.6; (2) it then encodes in SMT the conditions (9)–(13) defined through the paper, for each loop separately, by using template generation, a process that involves several non-trivial implementations using Farkas' lemma (this part is implemented in Java and uses Z3 [8] for simple (linear) satisfiability checks when producing the Max-SMT encoding); (3) the problem is solved using the SMT solver Barcelogic [4], as it allows us to use non-linear arithmetic and Max-SMT capabilities in order to assert soft conditions and implement the solutions described in Sect. 4; (4) in order to guarantee the correctness of our system results, we have added to the pipeline an additional checker that proves that the obtained metering function and quasi-invariants verify conditions (9)–(13) by using Z3. To empirically evaluate the results of our approach, we have used benchmarks from the Termination Problem Data Base (TPDB), namely those from the category *Complexity_ITS* that contains Integer Transition Systems. We have removed non-terminating TSs and terminating TSs whose cost is unbounded (i.e., the cost depends on some non-deterministic variables and can be arbitrarily high) or non-linear, because they are outside the scope of our approach. In total, we have considered a set of 473 multi-path loops from which we have excluded 13 that were non-linear. Analyzing these 473 programs took 199 min, an average of 25 sec by program, approximately. For 255 of them, it took less than 1 s.

Table 1 illustrates our results and compares them to those obtained by the LoAT [12,13] system, which also outputs a pair $(\rho, \mathcal{Q})$ of a lower-bound function $\rho$ and initial conditions $\mathcal{Q}$ on the input for which $\rho$ is a valid lower-bound. In order to automatically compare the results obtained by the two systems, we have implemented a comparator that first expresses costs as functions $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ over a single variable $n$ and then checks which function is greater. To obtain this unary cost function from the results $(\rho, \mathcal{Q})$, we use convex polyhedra manipulation libraries to maximize the obtained cost $\rho$ wrt. $\mathcal{Q} \wedge \overline{-n \leq x_i \leq n}$, where $x_i$ are the TS variables, and express that maximized expressions in terms of $n$. Therefore, $f(n)$ represents the maximum cost when the variables are bounded by $|x_i| \leq n$ and satisfy the corresponding initial condition $\mathcal{Q}$, a notion very similar to the runtime complexity used in [12,13]. Once we have both unary linear costs $f_1(n) = k_1 n + d_1$ and $f_2(n) = k_2 n + d_2$, we compare them in $n \geq 0$ by inspecting $k_1$ and $k_2$.

Each row of the table contains the number of loops for which both tools obtain the same result (=), the number of loops where LOBER is better than LoAT (>) and the number of loops where LoAT is better than LOBER (<). The subcategories are obtained directly from the name of the innermost folder, except for the cases in which this folder contains too few examples that we merge them

**Table 1.** Results of the experiments.

| Benchmark set | Total | = | > | < | Benchmark set | Total | = | > | < |
|---|---|---|---|---|---|---|---|---|---|
| BROCKSCHMIDT_16 | | | | | FGPSF09/Misc | 20 | 16 | 3 | 1 |
| c-examples/ABC | 33 | 33 | 0 | 0 | KoAT-2013 | 10 | 10 | 0 | 0 |
| c-examples/SPEED | 29 | 25 | 4 | 0 | KoAT-2014 | 14 | 14 | 0 | 0 |
| c-examples/WTC | 45 | 39 | 4 | 2 | SAS10 | 46 | 40 | 1 | 5 |
| c-examples/Misc | 9 | 9 | 0 | 0 | FLORES-MONTOYA_16 | 176 | 158 | 16 | 2 |
| costa | 6 | 5 | 1 | 0 | HARK_20 | | | | |
| FGPSF09/Beerendonk | 28 | 24 | 4 | 0 | Ben_Amram_Genaim | 10 | 7 | 2 | 1 |
| FGPSF09/patrs | 18 | 16 | 2 | 0 | Nils_2019 | 16 | 16 | 0 | 0 |

all in a Misc folder in the parent directory. The total number of loops that are considered in each subcategory appears in column **Total**. BROCKSCHMIDT_16 and HARK_20 have their first row empty as all their results are contained in their subcategories. Globally, both tools behave the same in 412 programs (column "="), obtaining equivalent linear lower bounds in 376 of them and a constant lower bound in the remaining ones. Our tool LOBER achieves a better accuracy in 37 programs (column ">"), while LoAT is more precise in 11 programs (column "<"). Let us discuss the two sets of programs in which both tools differ. As regards the 37 examples for which we get better results, we have that LoAT crashes in 4 cases and it can only find a constant lower bound in 1 example while our tool is able to find a path of linear length by introducing the necessary quasi-invariants. For the remaining 32 loops, both tools get a linear bound, but LOBER finds one that leads to an unboundedly longer execution: 18 of these loops correspond to cases that have implicit relations between the different execution paths (like our running examples) and require semantic reasoning; for the remaining 14, we get a better set of quasi-invariants. The following techniques have been needed to get such results in these 37 better cases (note that (i) is not mutually exclusive with the others):

(i) 1 needs narrowing non-deterministic choices,
(ii) 5 do not need quasi-invariants nor guard narrowing,
(iii) 14 need quasi-invariants only,
(iv) 18 need both quasi-invariants and guard narrowing (in 3 of them this is only used to disable transitions).

Therefore, this shows experimentally the relevance of all components within our framework and its practical applicability thanks to the good performance of the Max-SMT solver on non-linear arithmetic problems. In general, for all the set of programs, we can solve 308 examples without quasi-invariants and 444 without guard-narrowing. The intersection of these two sets is: 298 examples (63% of the programs), that leaves 175 programs that need the use of some of the proposed techniques to be solved.

As regards the 11 examples for which we get worse results than LoAT, we have two situations: (1) In 6 cases, the SMT-solver is not able to find a solution.

We noticed that too many quasi-invariants were required, what made the SMT problem too hard. To improve our results, we could start, as a preprocessing step, from a quasi-invariant that includes all invariant inequalities that syntactically appear in the loop transitions, something similar to what is done by LoAT when inferring what they call conditional metering function [12]. This is left for future experimentation. (2) In the other 5 cases, our tool finds a linear bound but with a worse set of quasi-invariants, which makes the LoAT bound provide unboundedly longer executions. We are investigating whether this can be improved by adding new soft constraints that guide the solver to find these better solutions. Finally, let us mention that, for the 13 problems that LoAT gives a non-linear bound and have been excluded from our benchmarks as justified above, we get a linear bound for the 12 that have a polynomial bound (of degree 2 or more), and a constant bound for the additional one that has a logarithmic lower bound. This is the best we can obtain as our approach focuses on the inference of precise local linear bounds, as they constitute the most common type of loops.

All in all, we argue that our experimental results are promising: we triple LoAT in the number of benchmarks for which we get more accurate results and, besides, many of those examples correspond to complex loops that lead to worse results when disconnecting transitions. Besides, we see room for further improvement, as most examples for which LoAT outperforms us could be handled as accurately as them with better quasi-invariants (that is somehow a black-box component in our framework). Syntactic strategies that use invariant inequalities that appear in the transitions, like those used in LoAT, would help, as well as further improvements in SMT non-linear arithmetic.

*Application Domains.* The accuracy gains obtained by LOBER have applications in several domains in which knowing the precise cost can be fundamental. This is the case for predicting the gas usage [26] of executing *smart contracts*, where gas cost amounts to monetary fees. The caller of a transaction needs to include a gas limit to run it. Giving a too low gas limit can end in an "out of gas" exception and giving a too high gas limit can end in a "not enough eth (money)" error. Therefore having a tighter prediction is needed to be safe on both sides. Also, when the UB is equal to the LB, we have an exact estimation, e.g., we would know precisely the runtime or memory consumption of the most costly executions. This can be crucial in safety-critical applications and has been used as well to detect potential vulnerabilities such as denial-of-service attacks. In https://apps.dtic.mil/sti/pdfs/AD1097796.pdf, vulnerabilities are detected in situations in which both bounds do not coincide. For instance, in password verification programs, if the UB and LB differ due to a difference on the delays associated to how many characters are right in the guessed password, this is identified as a potential attack.

## 6 Related Work and Conclusions

We have proposed a novel approach to synthesize precise lower-bounds from integer non-deterministic programs. The main novelties are on the use of loop specialization to facilitate the task of finding a (precise) metering function and on the Max-SMT encoding to find larger (better) solutions. Our work is related to two lines of research: (1) non-termination analysis and (2) LB inference. In both kinds of analysis, one aims at finding classes of inputs for which the program features a non-terminating behavior (1) or a cost-expensive behavior (2). Therefore, techniques developed for non-termination might provide a good basis for developing a LB analysis. In this sense, our work exploits ideas from the Max-SMT approach to non-termination in [17]. The main idea borrowed from [17] has been the use of quasi-invariants to specialize loops towards the desired behavior: in our case towards the search of a metering function, while in theirs towards the search of a non-termination proof. However, there are fundamental differences since we have proposed other new forms of loop specialization (see a more detailed comparison in Sect. 1) and have been able to adapt the use of Max-SMT to accurately solve our problem (i.e., find larger bounds). As mentioned in Sect. 1, our loop specialization technique can be used to gain precision in non-termination analysis [17]. For instance, in this loop: "while (x>=0 and y>=0) {if (∗) {x++; y−−;} else {x−−;y++;}}" no sub SCC (considering only one of the transitions) is non-terminating and no quasi-invariant can be found to ensure we will stay in the loop (when considering both transitions), hence cannot be handled by [17]. Instead if we narrow the transitions by adding $y >= x$ in the if-condition (and hence $x > y$ in the else), we can prove that $x >= 0 \wedge y >= 0 \wedge x + y = 1$ is quasi-invariant, which allow us to prove non-termination in the way of [17] (as we will stay in the loop forever).

As regards LB inference, the current state-of-the-art is the work by Frohn et al. [12,13] that introduces the notion of metering function and acceleration. Our work indeed tries to recover the semantic loss in [12,13] due to defining metering functions for simple loops and combining them in a later stage using acceleration. Technically, we only share with this work the basic definition of metering function in Sect. 3.1. Indeed, the definition in conditions (3) and (4) already generalizes the one in [12,13] since it is not restricted to simple loops. This definition is improved in the following sections with several loop specializations. While [12,13] relies on pure SMT to solve the problem, we propose to gain precision using Max-SMT. We believe that similar ideas could be adapted by [12,13]. Due to the different technical approaches underlying both frameworks, their accuracy and efficiency must be compared experimentally wrt. the LoAT system that implements the ideas in [12,13]. We argue that the results in Sect. 5 justify the important gains of using our new framework and prove experimentally that, the fact that we do not lose semantic relations in the search of metering functions is key to infer LB for challenging cases in which [12,13] fails. Originally, the LoAT [12,13] system only accelerated simple loops by using metering functions, so the overall precision of the lower bound relied on obtaining valid and precise metering functions. However, the framework in [12,13] is independent of the accelerating technique applied. In order to increase the number of simple loops that can be accelerated, Frohn [11] proposes a calculus to combine different conditional acceleration techniques (monotonic increase/decrease, eventual increase/decrease, and metering functions). These conditional acceleration techniques assume that all the iterations of the loop verify some condition $\varphi$, and the calculus applies the techniques in order and extract those conditions $\varphi$ from fragments of the loop guard. Although more precise and powerful, the combined acceleration calculus considers only simple loops, so it does not solve the precision loss when the loop cost involves several interleaved transitions. Moreover, the techniques in [11] are integrated into LoAT, so the experimental evaluation in Sect. 5 compares our approach to the framework in [12,13] extended with several techniques to accelerate loops (not only metering functions).

Finally, our approach presents similarities to the CTL* verification for ITS in [7] as both extend transition guards of the original ITS. The difference is that in [7] the added constraints only contain newly created *prophecy variables* and the transitions to modify are detected directly using graph algorithms; whereas our SMT-based approach adds constraints only over existing variables to satisfy the properties that characterize a good metering function. Additionally, both approaches differ both in the goal (CTL* verification vs. inference of lower-bounds) and the technologies applied (CTL model checkers vs. Max-SMT solvers).

## References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_15
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, Rocco (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_12
3. Albert, E., Genaim, S., Masud, A.N.: On the inference of resource usage upper and lower bounds. ACM Trans. Comput. Logic **14**(3), 1–35 (2013)
4. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The barcelogic SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_27
5. Brockschmidt, M., et al.: Analyzing runtime and size complexity of integer programs. ACM Trans. Program. Lang. Syst. **38**(4), 1–50 (2016)
6. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_39
7. Cook, B., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 13–29. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_2
8. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

9. Debray, S.K., López-García, P., Hermenegildo, M., Lin, N.-W.: Lower Bound Cost Estimation for Logic Programs. The MIT Press, In Logic Programming (1997)
10. Flores-Montoya, A.: Upper and lower amortized cost bounds of programs expressed as cost relations. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 254–273. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_16
11. Frohn., F.: A calculus for modular loop acceleration. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 58–76. Springer International Publishing (2020)
12. Frohn, F., Naaf, M., Brockschmidt, M., Giesl, J.: Inferring lower runtime bounds for integer programs. ACM Trans. Program. Lang. Syst. **42**(3), 1–50 (2020)
13. Frohn, F., Naaf, M., Hensel, J., Brockschmidt, M., Giesl, J.: Lower runtime bounds for integer programs. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 550–567. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_37
14. Gulwani, S., Mehra, K.K., Chilimbi, T.: SPEED. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2009. ACM Press (2008)
15. Hark, M., Kaminski, B.L., Giesl, J., Katoen, J.-P.: Aiming low is harder: induction for lower bounds in probabilistic program verification. In: Proceedings of the ACM on Programming Languages, 4(POPL), pp. 1–28, January 2020
16. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 132–157. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_6
17. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using Max-SMT. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 779–796. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_52
18. Ngo, V.C., Carbonneaux, Q., Hoffmann, J.: Bounded expectations: resource analysis for probabilistic programs. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, June 2018
19. Otto, C., Brockschmidt, M., Von Essen, C., Giesl, J.: Automated termination analysis of java bytecode by term rewriting. In: Lynch, C. (eds.) RTA, vol. 6 of LIPIcs, pp. 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
20. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)
21. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 745–761. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_50
22. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. J. Autom. Reasoning **59**(1), 3–45 (2017). https://doi.org/10.1007/s10817-016-9402-4
23. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. ACM SIGPLAN Not. **44**(6), 223–234 (2009)
24. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (1975)
25. Wei, Tao, Mao, Jian, Zou, Wei, Chen, Yu.: A new algorithm for identifying loops in decompilation. In: Nielson, Hanne Riis, Filé, Gilberto (eds.) SAS 2007. LNCS, vol. 4634, pp. 170–183. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_11
26. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2014)

# Fast Computation of Strong Control Dependencies

Marek Chalupa(✉), David Klaška, Jan Strejček, and Lukáš Tomovič

Masaryk University, Brno, Czech Republic
{chalupa,strejcek}@fi.muni.cz,
{david.klaska,tomovic}@mail.muni.cz

**Abstract.** We introduce new algorithms for computing *non-termination sensitive control dependence* (NTSCD) and *decisive order dependence* (DOD). These relations on vertices of a control flow graph have many applications including program slicing and compiler optimizations. Our algorithms are asymptotically faster than the current algorithms. We also show that the original algorithms for computing NTSCD and DOD may produce incorrect results. We implemented the new as well as fixed versions of the original algorithms for the computation of NTSCD and DOD. Experimental evaluation shows that our algorithms dramatically outperform the original ones.

## 1 Introduction

Control dependencies between program statements are studied since 70's. They have important applications in compiler optimizations [12,14,16], program analysis [9,19,36], and program transformations, especially program slicing [1,9,22,26,37]. Slicing is used in many areas including testing, debugging, parallelization, reverse engineering, program analysis and verification [17,28].

Informally, two statements in a program are control dependent if one directly controls the execution of the other in some way. This is typically the case for **if** statements and their bodies. Control dependencies are nowadays classified as *weak (non-termination insensitive)* if they assume that a given program always terminates, or as *strong (non-termination sensitive)* if they do not have this assumption [13]. We illustrate the difference on the control flow graph in Fig. 1. Node $a$ controls whether $b$ or $c$ (and then $d$) is going to be executed, so $b$, $c$, and $d$ are control dependent on $a$ (the convention is to display dependence as edges in the "controls" direction). Similarly, $b$ controls the execution of $c$ and $d$, as these nodes may be bypassed by going from $b$ to $e$. Note also that $d$ controls whether $d$ is going to be executed in the future and thus is control dependent on itself. However, $c$ does not control $d$ as any path from $c$ hits $d$. All dependencies mentioned so far are weak, namely *standard control dependencies* as defined by Ferrante et al. [16]. Weak control dependence assumes that the program always terminates, in particular, that the loop over $d$ cannot iterate forever. As a result,

© The Author(s) 2021
A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 887–910, 2021.
https://doi.org/10.1007/978-3-030-81688-9_41

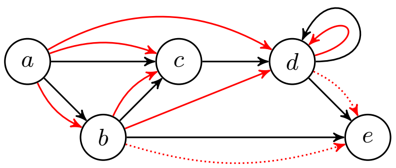


**Fig. 1.** An example of a control flow graph and control dependencies (red edges). The dotted dependencies are additional non-termination sensitive control dependencies. (Color figure online)

$e$ is reached by all executions and thus it is not weakly control dependent on any node. However, $e$ is strongly control dependent on $b$ and $d$. Indeed, if we assume that some executions can loop over $d$ forever, then reaching $e$ is controlled clearly by $d$ and also by $b$ as it can send the execution directly to $e$.

This paper is concerned with the computation of two prominent strong control dependencies introduced by Ranganath et al. [32,33], namely *non-termination sensitive control dependence (NTSCD)* and *decisive order dependence (DOD)*. NTSCD is studied in Sect. 3, which follows after preliminaries in Sect. 2. We first recall the definition of NTSCD and the algorithm of Ranganath et al. [33] for its computation. Then we show a flaw in the algorithm and suggest a fix. Finally, we introduce a new algorithm for the computation of NTSCD. Given a control flow graph with $|V|$ nodes, the new algorithm runs in time $O(|V|^2)$, while the algorithm of Ranganath et al. runs in time $O(|V|^4 \cdot \log |V|)$ and its fixed version in time $O(|V|^5)$. We show a NTSCD relation of size $\Theta(|V|^2)$, which means that our algorithm is asymptotically optimal.

The DOD relation captures the cases when one node controls the execution order of two other nodes. Roughly speaking, nodes $\{b, c\}$ are DOD on $a$ whenever all executions passing through $a$ eventually reach both $b$ and $c$ and $a$ controls which is reached first. Ranganath et al. [33] proved that the relation is empty for *reducible* graphs [21], i.e., graphs where every cycle has a single entry point. Control flow graphs of structured programs are reducible, but irreducible graphs may arise for example in the following situations [11,33,35]:

- unstructured coding by a human, which is rather rare nowadays,
- compilation into unstructured code representation like JVM bytecode,
- tail call recursion optimization during compilation,
- when the control flow graph is interprocedural – in this case, irreducibility may be introduced by recursion or exceptions handling,
- by reversing a control flow graph containing, for example, break statements
- when the control flow graph is not generated from program, but, e.g., from a finite state machine.

The DOD relation is important (together with NTSCD) when we want to slice possibly non-terminating programs with irreducible control flow graphs and preserve their termination properties as well as data integrity [1,33]. This is a common requirement when slicing is used as a preprocessing step before program verification [9,23,26], worst-case execution time analysis [29], information flow analysis [18,19], analysis of concurrent programs [18] with busy-waiting synchronization or synchronization where possible spurious wake-ups of threads are guarded by loops (e.g., programs using the *pthread* library), and analysis of reactive systems and generic state-based models [2,24,33].

The DOD relation is studied in Sect. 4, where we recall its definition, discuss the Ranganath et al.'s algorithm for DOD [33], and show that this algorithm also contains a flaw. Fortunately, this flaw can be easily fixed without changing the complexity of the algorithm. Further, we develop a theory that underpins our new algorithm for the computation of DOD. Due to the space limitations, proofs of theorems can be found only in the extended version of this paper [8]. The new algorithm, presented at the end of the section, computes DOD in time $O(|V|^3)$, while the original as well as the fixed version of the Ranganath et al.'s algorithm runs in $O(|V|^5 \cdot \log |V|)$. We show a DOD relation of size $\Theta(|V|^3)$, which means that our algorithm is again asymptotically optimal.

Section 5 focuses on *control closures (CC)* introduced by Danicic et al. [33], which generalize control dependence to arbitrary directed graphs. It is known that the *strong* (i.e., non-termination sensitive) control closure for a set of nodes containing the starting node is equivalent to the closure under NTSCD and DOD relations. Hence, our algorithms for NTSCD and DOD can be used to compute strong CC in time $O(|V|^3)$ on control flow graphs, while the original algorithm by Danicic et al. [13] runs in $O(|V|^4)$.

Our theoretical contribution to computation of strong control dependencies is summarized in Table 1. Section 6 presents experimental evaluation showing that our algorithms are indeed dramatically faster than the original ones. The paper is concluded with Sect. 7.

## 1.1 Related Work

The first paper concerned with control dependence is due to Denning and Denning [15], who used control dependence to certify that flow of information in a program is secure. Weiser [37], Ottenstein and Ottenstein [30], and Ferrante et al. [16] used control dependence in program slicing, which is also the motivation for the most of the latter research in this area. These "classical" papers study control dependence in terminating programs with a unique exit node eventually reached by every execution. These restrictions have been gradually removed.

**Table 1.** Overview of discussed algorithms and their complexities on CFGs

| Relation/closure | Algorithm | Complexity |
|---|---|---|
| NTSCD | Original algorithm by Ranganath et al. [33] | $O(\|V\|^4 \cdot \log \|V\|)$ |
| (Sect. 3) | Fixed algorithm by Ranganath et al. [33] | $O(\|V\|^5)$ |
| | New algorithm | $O(\|V\|^2)$ |
| DOD | Original algorithm by Ranganath et al. [33] | $O(\|V\|^5 \cdot \log \|V\|)$ |
| (Sect. 4) | Fixed algorithm by Ranganath et al. [33] | $O(\|V\|^5 \cdot \log \|V\|)$ |
| | New algorithm | $O(\|V\|^3)$ |
| Strong CC | Original algorithm by Danicic et al. [13] | $O(\|V\|^4)$ |
| (Sect. 5) | New NTSCD-and-DOD-based algorithm | $O(\|V\|^3)$ |

Podgurski and Clarke [31] defined the first strong control dependence that does not assume termination of the program.[1] However, their definitions and algorithms still require programs to have a unique exit node.

Bilardi and Pingal [5] introduced a framework that uses generalized dominance relation on graphs. In their framework, they are able to compute Podgurski and Clarke's control dependence in $O(|E|+|V|^2)$ time for a directed graph $(V, E)$ with a unique exit node. In theory, NTSCD could be computed in their framework. However, computing *augmented post-dominator tree* – the central data structure of their framework – requires the unique exit node as it starts with post-dominator tree and, mainly, is much more complicated compared to our algorithm for NTSCD [5].

Chen and Rosu [10] introduced a parametric approach where loops can be annotated with information about termination. The resulting control dependence is somewhere between the classical and Podgurski and Clarke's control dependence, the two being the extremes.

The notion of NTSCD and DOD was founded in works of Ranganath et al. [32,33] in order to slice reactive systems, e.g., operating systems or controllers of embedded devices. They generalized also classical (*non-termination insensitive*) control dependence to graphs without the unique exit point (further investigated, e.g., by Androutsopoulos et al. [3]) and provided several relaxed versions of DOD.

Danicic et al. [13] introduced *weak* and *strong control closures (CC)* that generalize weak and strong control dependence (thus also NTSCD) to arbitrary graphs. They provide algorithms for the computation of minimal closures that run in $O(|V|^3)$ (weak CC) and $O(|V|^4)$ (strong CC) on graph with $|V|$ nodes.

An orthogonal study of control dependence that arises between statements in different procedures (e.g., due to calls to `exit()`) was carried out by Loyall and Mathisen [27], Harrold et al. [20], and Sinha et al. [34].

## 2 Preliminaries

A *finite directed graph* is a pair $G = (V, E)$, where $V$ is a finite set of *nodes* and $E \subseteq V \times V$ is a set of *edges*. If there is an edge $(m, n) \in E$, then $n$ is called a *successor* of $m$, $m$ is a *predecessor* of $n$, and the edge is an *outgoing edge* of $m$. Given a node $n$, *Successors*$(n)$ and *Predecessors*$(n)$ denote the sets of all its successors and predecessors, respectively. A *path* from a node $n_1$ is a nonempty finite or infinite sequence $n_1 n_2 \ldots \in V^+ \cup V^\omega$ of nodes such that there is an edge $(n_i, n_{i+1}) \in E$ for each pair $n_i, n_{i+1}$ of adjacent nodes in the sequence. A path is called *maximal* if it cannot be prolonged, i.e., it is infinite or the last node of the path has no outgoing edge. A node $m$ is *reachable* from a node $n$ if there exists a finite path such that its first node is $n$ and its last node is $m$.

We say that a graph is a *cycle*, if it is isomorphic to a graph $(V, E)$ where $V = \{n_1, \ldots, n_k\}$ for some $k > 0$ and $E = \{(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k),$

[1] Podgurski and Clarke [31] called their control dependence *weak control dependence* as it is a superset of classical control dependence. Nowadays, we use the terms *weak* and *strong* precisely in the opposite meaning [13].

$(n_k, n_1)\}$. A cycle *unfolding* is a path in the cycle that contains each node precisely once.

In this paper, we consider programs represented by control flow graphs, where nodes correspond to program statements and edges model the flow of control between the statements. As control dependence reflects only the program structure, our definition of a control flow graph does not contain any statements. Our definition also does not contain any start or exit nodes as these are not important for the problems we study in this paper.

**Definition 1 (Control flow graph, CFG).** *A* control flow graph (CFG) *is a finite directed graph $G = (V, E)$ where each node $v \in V$ has at most two outgoing edges. Nodes with exactly two outgoing edges are called* predicate nodes *or simply* predicates*. The set of all predicates of a CFG $G$ is denoted by $Predicates(G)$.*

## 3 Non-termination Sensitive Control Dependence

This section recalls the definition of NTSC by Ranganath et al. [32] and their algorithm for computing NTSCD. Then we show that the algorithm can produce incorrect results and introduce a new algorithm that is asymptotically faster.

**Definition 2 (Non-termination sensitive control dependence, NTSCD).** *Given a CFG $G = (V, E)$, a node $n \in V$ is* non-termination sensitive control dependent (NTSCD) *on a predicate node $p \in Predicates(G)$, written $p \xrightarrow{NTSCD} n$, if $p$ has two successors $s_1$ and $s_2$ such that*

- *all maximal paths from $s_1$ contain $n$, and*
- *there exists a maximal path from $s_2$ that does not contain $n$.*

### 3.1 Algorithm of Ranganath et al. [33] for NTSCD

The algorithm is presented in Algorithm 1. Its central data structure is a two-dimensional array $S$ where for each node $n$ and for each predicate node $p$ with successors $r$ and $s$, $S[n, p]$ always contains a subset of $\{t_{pr}, t_{ps}\}$. Intuitively, $t_{pr}$ should be added to $S[n, p]$ if $n$ appears on all maximal paths from $p$ that start with the prefix $pr$. The *workbag* holds the set of nodes $n$ for which some $S[n, p]$ value has been changed and this change should be propagated. The first part of the algorithm initializes the array $S$ with the information that each successor $r$ of a predicate node $p$ is on all maximal paths from $p$ starting with $pr$. The main part of the algorithm then spreads the information about the reachability on all maximal paths in the forward manner. Finally, the last part computes the NTSCD relation according to Definition 2 and with use of the information in $S$.

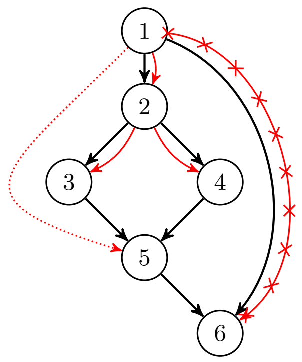
The algorithm runs in time $O(|E| \cdot |V|^3 \cdot \log |V|)$ [33] for a CFG $G = (V, E)$. The $\log |V|$ factor comes from set operations. Since every node in CFG has at most 2 outgoing edges, we can simplify the complexity to $O(|V|^4 \cdot \log |V|)$.

Although the correctness of the algorithm has been proved [32, Theorem 7], Fig. 2 presents an example where the algorithm provides an incorrect answer.

**Algorithm 1:** The NTSCD algorithm by Ranganath et al. [33]

```
    Input: a CFG $G = (V, E)$
    Output: a potentially incorrect NTSCD relation stored in $ntscd$
 1  Set $S[n,p] = \emptyset$ for all $n \in V$ and $p \in Predicates(G)$      // Initialization
 2  $workbag \leftarrow \emptyset$
 3  for $p \in Predicates(G)$ do
 4      for $r \in Successors(p)$ do
 5          $S[r,p] \leftarrow \{t_{pr}\}$
 6          $workbag \leftarrow workbag \cup \{r\}$
 7
 8  while $workbag \neq \emptyset$ do                                    // Computation of $S$
 9      $n \leftarrow$ pop from $workbag$
10      if $Successors(n) = \{s\}$ for some $s \neq n$ then          // One successor case
11          for $p \in Predicates(G)$ do
12              if $S[n,p] \smallsetminus S[s,p] \neq \emptyset$ then
13                  $S[s,p] \leftarrow S[s,p] \cup S[n,p]$
14                  $workbag \leftarrow workbag \cup \{s\}$
15      if $|Successors(n)| > 1$ then                            // Multiple successors case
16          for $m \in V$ do
17              if $|S[m,n]| = |Successors(n)|$ then
18                  for $p \in Predicates(G) \smallsetminus \{n\}$ do
19                      if $S[n,p] \smallsetminus S[m,p] \neq \emptyset$ then
20                          $S[m,p] \leftarrow S[m,p] \cup S[n,p]$
21                          $workbag \leftarrow workbag \cup \{m\}$
22
23  $ntscd \leftarrow \emptyset$                                          // Computation of NTSCD
24  for $n \in V$ do
25      for $p \in Predicates(G)$ do
26          if $0 < |S[n,p]| < |Successors(p)|$ then
27              $ntscd \leftarrow ntscd \cup \{p \xrightarrow{\text{NTSCD}} n\}$
```

The first part of the algorithm initializes $S$ as shown in the figure and sets *workbag* to $\{2, 6, 3, 4\}$. Then any node from *workbag* can be popped and processed. Let us apply the policy used for queues: always pop the oldest element in *workbag*. Hence, we pop 2 and nothing happens as the condition on line 17 is not satisfied for any $m$. This also means that the symbol $t_{12}$ is not propagated any further. Next we pop 6, which has no effect as 6 has no successor. By processing 3 and 4, $t_{23}$ and $t_{24}$ are propagated to $S[5, 2]$ and 5 is added to the *workbag*. Finally, we process 5 and set $S[6, 2]$ to $\{t_{23}, t_{24}\}$. The final content of $S$ is provided in the figure. Unfortunately, the information in $S$ is sound but incomplete. In other words, if $t_{pr} \in S[n,p]$, then $n$ is indeed on all maximal paths from $p$ starting with $pr$, but the opposite implication does not hold. In particular, $t_{12}$ is missing in $S[5, 1]$ and $S[6, 1]$. Consequently, the last part of the algorithm computes an incorrect NTSCD relation: it correctly identifies $1 \xrightarrow{\text{NTSCD}} 2$, $2 \xrightarrow{\text{NTSCD}} 3$, and $2 \xrightarrow{\text{NTSCD}} 4$, but it also incorrectly produces $1 \xrightarrow{\text{NTSCD}} 6$ and misses $1 \xrightarrow{\text{NTSCD}} 5$.



| $S$ after initialization |
|---|
| $S[2,1] = \{t_{12}\}$ |
| $S[6,1] = \{t_{16}\}$ |
| $S[3,2] = \{t_{23}\}$ |
| $S[4,2] = \{t_{24}\}$ |

| final $S$ when nodes are popped in order | |
|---|---|
| $2,6,3,4,5$ (oldest first) | $3,4,2,5,6$ (correct) |
| $S[2,1] = \{t_{12}\}$ | $S[2,1] = \{t_{12}\}$ |
| $S[6,1] = \{t_{16}\}$ | $S[3,2] = \{t_{23}\}$ |
| $S[3,2] = \{t_{23}\}$ | $S[4,2] = \{t_{24}\}$ |
| $S[5,2] = \{t_{23}, t_{24}\}$ | $S[5,1] = \{t_{12}\}$ |
| $S[6,2] = \{t_{23}, t_{24}\}$ | $S[6,1] = \{t_{12}, t_{16}\}$ |
| | $S[6,2] = \{t_{23}, t_{24}\}$ |

**Fig. 2.** An example that shows the incorrectness of the NTSCD algorithm by Ranganath et al. [33]. Solid red edges depict the dependencies computed by the algorithm when it always pops the oldest element in *workbag*. The crossed dependence is incorrect. The dotted dependence is missing in the result.

A necessary condition to get the correct result is to process 2 only after $3, 4$ are processed and $S[5,6] = \{t_{23}, t_{24}\}$. For example, one obtains the correct $S$ (also shown in the figure) when the nodes are processed in the order $3, 4, 2, 5, 6$.

The algorithm is clearly sensitive to the order of popping nodes from *workbag*. We are currently not sure whether for each *CFG* there exists an order that leads to the correct result. An easy way to fix the algorithm is to ignore the *workbag* and repeatedly execute the body of the **while** loop (lines 10–21) for all $n \in V$ until the array $S$ reaches a fixpoint. However, this modification would slow down the algorithm substantially. Computing the fixpoint needs $O(|V|^3)$ iterations over the loop body (lines 10–21 excluding lines 14 and 21 handling the *workbag*) and one iteration of this loop body needs $O(|V|^2)$. Hence, the overall time complexity of the fixed version is $O(|V|^5)$.

## 3.2 New Algorithm for NTSCD

We have designed and implemented a new algorithm computing NTSCD. Our algorithm is correct, significantly simpler and asymptotically faster than the original algorithm of Ranganath et al. [33].

The new algorithm calls for each node $n$ a procedure that identifies all NTSCD dependencies of $n$ on predicate nodes. The procedure works in the following steps.

1. Color $n$ red.
2. Pick an uncolored node such that it has some successors and they all are red. Color the node red. Repeat this step until no such node exists.
3. For each predicate node $p$ that has a red successor and an uncolored one, output $p \xrightarrow{\text{NTSCD}} n$.

**Algorithm 2:** The new NTSCD algorithm

```
Input: a CFG $G = (V, E)$
Output: the NTSCD relation stored in $ntscd$

Procedure VISIT($n$)                                  // Auxiliary procedure
    $n.counter \leftarrow n.counter - 1$
    if $n.counter = 0 \ \wedge \ n.color \neq red$ then
        $n.color \leftarrow red$
        for $m \in Predecessors(n)$ do
            VISIT($m$)

Procedure COMPUTE($n$)          // Coloring the graph red for a given $n$
    for $m \in V$ do
        $m.color \leftarrow uncolored$
        $m.counter \leftarrow |Successors(m)|$
    $n.color \leftarrow red$
    for $m \in Predecessors(n)$ do
        VISIT($m$)

$ntscd \leftarrow \emptyset$                                   // Computation of NTSCD
for $n \in V$ do
    COMPUTE($n$)
    for $p \in Predicates(G)$ do
        if $p$ has a red successor and an uncolored successor then
            $ntscd \leftarrow ntscd \cup \{p \xrightarrow{\text{NTSCD}} n\}$
```

Unlike the Ranganath et al.'s algorithm which works in a forward manner, our algorithm spreads the information about the reachability of $n$ on all maximal paths in the backward direction starting from $n$.

The algorithm is presented in Algorithm 2. The procedure COMPUTE($n$) implements the first two steps mentioned above. In the second step, it does not search over all nodes to pick the next node to color. Instead, it maintains the count of uncolored successors for each node. Once the count drops to 0 for a node, the node is colored red and the search continues with predecessors of this node. The third step is implemented directly in the main loop of the algorithm.

To prove that the algorithm is correct, we basically need to show that when COMPUTE($n$) finishes, a node $m$ is red iff all maximal paths from $m$ contain $n$. We start with a simple observation.

**Lemma 1.** *After* COMPUTE($n$) *finishes, a node $m$ is red if and only if $m = n$ or $m$ has a positive number of successors and all of them are red.*

*Proof.* For each node $m$, the counter is initialized to the number of its successors and it is decreased by calls to VISIT($m$) each time a successor of $m$ gets red. When the counter drops to 0 (i.e., all successors of the node are red), the node is colored red. Therefore, if $m$ is red, it got red either on line 12 and $m = n$, or $m \neq n$ and $m$ is red because all its successors got red (it must have a positive number of successors, otherwise the counter could not be 0 after its decrement). In the other direction, if $m = n$, it gets red on line 12. If it has a positive number of successors which all get red, the node is colored red by the argument above. □

**Theorem 1.** *After* COMPUTE($n$) *finishes, for each node $m$ it holds that $m$ is red if and only if all maximal paths from $m$ contain $n$.*

*Proof.* ("$\Longleftarrow$") We prove this implication by contraposition. Assume that $m$ is an uncolored node. Lemma 1 implies that each uncolored node has an uncolored successor (if it has any). Hence, we can construct a maximal path from $m$ containing only uncolored nodes simply by always going to an uncolored successor, either up to infinity or up to a node with no successors. This uncolored maximal path cannot contain $n$ which is red.

("$\Longrightarrow$") For the sake of contradiction, assume that there is a red node $m$ and a maximal path from $m$ that does not contain $n$. Lemma 1 implies that all nodes on this path are red. If the maximal path is finite, it has to end with a node without any successor. Lemma 1 says that such a node can be red if and only if it is $n$, which is a contradiction. If the maximal path is infinite, it must contain a cycle since the graph is finite. Let $r$ be the node on this cycle that has been colored red as the first one. Let $s$ be the successor of $r$ on the cycle. Recall that $r \neq n$ as the maximal path does not contain $n$. Hence, node $r$ could be colored red only when all its successors including $s$ were already red. This contradicts the fact that $r$ was colored red as the first node on the cycle. □

To determine the complexity of our algorithm on a CFG $(V, E)$, we first analyze the complexity of one run of COMPUTE($n$). The lines 9–11 iterate over all nodes. The crucial observation is that the procedure VISIT is called at most once for each edge $(m, m') \in E$ of the graph: to decrease the counter of $m$ when $m'$ gets red. Hence, the procedure COMPUTE($n$) runs in $O(|V| + |E|)$. This procedure is called on line 18 for each node $n$. Finally, lines 20–21 are executed for each pair of node $n$ and predicate node $p$. This gives us the overall complexity $O((|V| + |E|) \cdot |V| + |V|^2) = O((|V| + |E|) \cdot |V|)$. Since in control flow graphs it holds $|E| \leq 2|V|$, the complexity can be simplified to $O(|V|^2)$.

Note that our algorithm is asymptotically optimal as there are CFGs with NTSCD relations of size $\Theta(|V|^2)$. For example, the CFG in Fig. 3 has $|V| = 2k+1$ nodes and the corresponding NTSCD relation

$$\{n_i \xrightarrow{\mathsf{NTSCD}} m_j \mid i, j \in \{1, \ldots, k\}\} \ \cup \ \{n_i \xrightarrow{\mathsf{NTSCD}} n_{i+1} \mid i \in \{1, \ldots, k-1\}\}$$

is of size $k^2 + k - 1 \in \Theta(|V|^2)$.

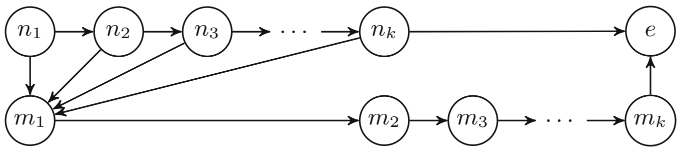


**Fig. 3.** A CFG with $|V|$ nodes that has the NTSCD relation of size $\Theta(|V|^2)$.

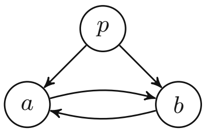


**Fig. 4.** An example of an irreducible CFG. There are no NTSCD dependencies, but $a$ and $b$ are DOD on $p$.

## 4 Decisive Order Dependence

There are control dependencies not captured by NTSCD. For example, consider the CFG in Fig. 4. Nodes $a$ and $b$ are not NTSCD on $p$ as they lie on all maximal paths from $p$. However, $p$ controls which of $a$ and $b$ is executed first. Ranganath et al. [33] introduced the DOD relation to capture such dependencies.

**Definition 3 (Decisive order dependence, DOD).** *Let $G = (V, E)$ be a CFG and $p, a, b \in V$ be three distinct nodes such that $p$ is a predicate node with successors $s_1$ and $s_2$. Nodes $a, b$ are* decisive order-dependent (DOD) *on $p$, written $p \xrightarrow{\text{DOD}} \{a, b\}$, if*

- *all maximal paths from $p$ contain both $a$ and $b$,*
- *all maximal paths from $s_1$ contain $a$ before any occurrence of $b$, and*
- *all maximal paths from $s_2$ contain $b$ before any occurrence of $a$.*

The importance of DOD for slicing of irreducible programs is discussed in the introduction.

### 4.1 Algorithm of Ranganath et al. [33] for DOD

Ranganath et al. provided an algorithm that computes the DOD relation for a given CFG $G = (V, E)$ in time $O(|V|^4 \cdot |E| \cdot \log |V|)$ which amounts to $O(|V|^5 \cdot \log |V|)$ on CFGs [33, Fig. 7]. The algorithm contains one unclear point. For each triple of nodes $p, a, b \in V$ such that $p \in \mathit{Predicates}(G)$ and $a \neq b$, the algorithm executes the following check and if it succeeds, then $p \xrightarrow{\text{DOD}} \{a, b\}$ is reported:

$$\text{REACHABLE}(a, b, G) \wedge \text{REACHABLE}(b, a, G) \wedge \text{DEPENDENCE}(p, a, b, G) \qquad (1)$$

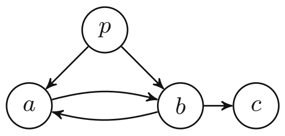


**Fig. 5.** An example that shows the incorrectness of the DOD algorithm by Ranganath et al. [33]

The procedure DEPENDENCE$(p, a, b, G)$ returns *true* iff $a$ is on all maximal paths from one successor of $p$ before any occurrence of $b$ and $b$ is on all maximal paths from the other successor of $p$ before any occurrence of $a$. The procedure REACHABLE is specified only by words [33, description of Fig. 7] as follows:

REACHABLE$(a, b, G)$ returns *true* if $b$ is reachable from $a$ in the graph $G$.

Unfortunately, this algorithm can provide incorrect results. For example, consider the CFG in Fig. 5. Nodes $p, a, b$ satisfy the formula (1): $a$ appears on all maximal paths from one successor of $p$ (namely $a$) before any occurrence of $b$, and $b$ appears on all maximal paths from the other successor of $p$ (which is $b$) before any occurrence of $a$. At the same time, $a$ and $b$ are reachable from each other. However, it is not true that $p \xrightarrow{\text{DOD}} \{a, b\}$, because $a$ and $b$ do not lie on all maximal paths from $p$ (the first condition of Definition 3 is violated).

The algorithm can be fixed by modifying the procedure REACHABLE$(a, b, G)$ to return *true* if $b$ is on all maximal paths from $a$. The modified procedure can be implemented with use of the procedure COMPUTE$(b)$ of Algorithm 2. As the procedure COMPUTE$(b)$ runs in $O(|V| + |E|)$, the modification does not increase the overall complexity of the algorithm. By comparing the fixed and the original version of REACHABLE$(a, b, G)$, one can readily confirm that the original version produces supersets of DOD relations.

## 4.2 New Algorithm for DOD: Crucial Observations

As in the case of NTSCD, we have designed a new algorithm for the computation of DOD, which is relatively simple and asymptotically faster than the DOD algorithm of Ranganath et al. [33].

Given a CFG, our algorithm first computes for each predicate $p$ the set $V_p$ of nodes that are on all maximal paths from $p$. The definition of DOD implies that only pairs of nodes in $V_p$ can be DOD on $p$. For every predicate $p$ we build an auxiliary graph $A_p$ with nodes $V_p$ and from this graph we get all pairs of nodes that are DOD on $p$. The graph $A_p$ is defined as follows.

**Definition 4 ($V'$-interval [13]).** *Given a CFG $G = (V, E)$ and a subset $V' \subseteq V$, a path $n_1 \ldots n_k$ such that $k \geq 2$, $n_1, n_k \in V'$, and $\forall 1 < i < k : n_i \notin V'$ is called a $V'$-*interval *from $n_1$ to $n_k$ in $G$.*

In other words, a $V'$-interval is a finite path with at least one edge that has the first and the last node in $V'$ but no other node on the path is in $V'$.

**Definition 5 (Graph $A_p$[2]).** *Given a CFG $G = (V, E)$, a predicate node $p \in$ Predicates$(G)$ and the subset $V_p \subseteq V$ of nodes that are on all maximal paths from $p$, the $A_p = (V_p, E_p)$ is the graph where*

$$E_p = \{(x, y) \mid \textit{there exists a } V_p\textit{-interval from } x \textit{ to } y \textit{ in } G\}.$$

In this subsection, we describe the connections between these graphs and DOD that underpin our algorithm. The proofs of the theorems can be found in the extended version of this paper [8].

Given a predicate $p$ of a CFG $G$, the graph $A_p$ does not have to be a CFG as nodes in $A_p$ can have more than two successors. However, $A_p$ preserves exactly all possible orders of the first occurrences of nodes in $V_p$ on maximal paths in $G$ starting from $p$. More precisely, for each maximal path from $p$ in $G$, there exists a maximal path from $p$ in $A_p$ with the same order of the first occurrences of all nodes in $V_p$, and vice versa. Further, it turns out that there are no nodes DOD on $p$ unless $A_p$ has the *right shape.*

**Definition 6 (Right shape of $A_p$ ).** *Given a CFG $G$, a predicate node $p \in$ Predicates$(G)$ and the graph $A_p = (V_p, E_p)$, we say that $A_p$ has the* right shape *if it consists only of a cycle and the node $p$ with at least two edges going to some nodes on the cycle (i.e., the nodes of $V_p \smallsetminus \{p\}$ can be labeled $n_1, \ldots, n_k$ such that $E_p = \{(n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k), (n_k, n_1)\} \cup \{(p, n_i) \mid i \in I\}$ for some $I \subseteq \{1, \ldots, k\}$ with $|I| \geq 2$).*

Figure 6 depicts an $A_p$ which has the right shape. In the following text, we work only with $A_p$ graphs in the right shape.

Let $s_1$ and $s_2$ be the two successors of $p$ in $G$. Note that $s_1$ and $s_2$ may, but do not have to be in $A_p$. To compute the pairs of nodes that are DOD on $p$, we need to know all possible orders of the first occurrences of nodes in $V_p$ on the maximal paths in $G$ starting in $s_1$ and $s_2$. Hence, for each successor $s_i$ we compute the set $S_i$ of nodes that appear as the first node of $V_p$ on some maximal path from $s_i$ in $G$. Formally, for $i \in \{1, 2\}$, we define

$$S_i = \{n \in V_p \mid \text{there exists a path } s_i \ldots n \in (V \smallsetminus V_p)^*.V_p \text{ in } G\}.$$

The nodes in $S_1 \cup S_2$ are exactly all the successors of $p$ in $A_p$. Further, the maximal paths from the nodes of $S_i$ in $A_p$ reflect exactly all possible orders of the first occurrences of nodes in $V_p$ on maximal paths in $G$ starting in $s_i$. If $S_1$ and $S_2$ are not disjoint, then there exist two maximal paths in $G$, one starting in $s_1$ and the other in $s_2$, that differ only in prefixes of nodes outside $V_p$. The definition of DOD implies that there are no nodes DOD on $p$ in this case. Therefore we assume that $S_1$ and $S_2$ are disjoint.

The nodes in $S_i$ divide the cycle of $A_p$ into $s_i$*-strips*, which are parts of the cycle starting with a node from $S_i$ and ending before the next node of $S_i$.

[2] Graph $A_p$ can be defined as the graph induced by $V_p$ in terms of Danicic et al. [13].

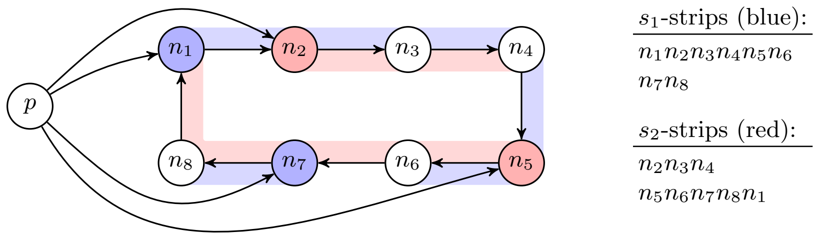


**Fig. 6.** An example of $A_p$ in the right shape. Strips are computed for $S_1 = \{n_1, n_7\}$ (blue nodes) and $S_2 = \{n_2, n_5\}$ (red nodes). (Color figure online)

**Definition 7 ($s_i$-strip).** *Let $i \in \{1, 2\}$. An* $s_i$-strip *is a path $n \dots m \in S_i.(V_p \setminus S_i)^*$ in $A_p$ such that the successor of $m$ in $A_p$ is a node in $S_i$.*

An example of $A_p$ with $s_i$-strips is in Fig. 6. The $s_i$-strips directly say which pairs of nodes of $V_p$ are in the same order on all maximal paths from $s_i$ in $G$. In particular, a node $a$ is before any occurrence of node $b$ on all maximal paths from a successor $s$ of $p$ in $G$ if and only if there is an $s$-strip containing both $a$ and $b$ where $a$ is before $b$. As a corollary, we get the following theorem:

**Theorem 2.** *Let $p$ be a predicate with successors $s_1, s_2$ such that $A_p$ has the right shape and $S_1 \cap S_2 = \emptyset$. Then nodes $a, b \in V_p$ are DOD on $p$ if and only if*

- *there exists an $s_1$-strip in $A_p$ that contains $a$ before $b$ and*
- *there exists an $s_2$-strip in $A_p$ that contains $b$ before $a$.*

Consider again the $A_p$ in Fig. 6. The theorem implies that nodes $n_1, n_5$ are DOD on $p$ as they appear in $s_1$-strip $n_1n_2n_3n_4n_5n_6$ and in $s_2$-strip $n_5n_6n_7n_8n_1$ in the opposite order. Nodes $n_1, n_6$ are DOD on $p$ for the same reason.

With use of the previous theorem, we can find a regular language over $V_p$ such that there exist nodes $a, b$ DOD on $p$ iff some unfolding of the cycle in $A_p$ is in the language.

**Theorem 3.** *Let $p$ be a predicate with successors $s_1, s_2$ such that $A_p$ has the right shape and $S_1 \cap S_2 = \emptyset$. Further, let $U = V_p \setminus (S_1 \cup S_2)$. There are some nodes $a, b$ DOD on $p$ if and only if the cycle in $A_p$ has an unfolding of the form $S_1.U^*.(S_2.U^*)^*.S_2.U^*.(S_1.U^*)^*$.*

Finally, an unfolding of the mentioned form can be directly used for the computation of nodes that are DOD on $p$.

**Theorem 4.** *Let $p$ be a predicate with successors $s_1, s_2$ such that $A_p$ has the right shape and $S_1 \cap S_2 = \emptyset$. Further, let $A_p$ have an unfolding of the form $S_1.U^*.(S_2.U^*)^*.S_2.U^*.(S_1.U^*)^*$ where $U = V_p \setminus (S_1 \cup S_2)$. Then there is exactly one path $m_1 \dots m_i \in S_1.U^*.S_2$ and exactly one path $o_1 \dots o_j \in S_2.U^*.S_1$ on the cycle. Moreover, $p \xrightarrow{DOD} \{a, b\}$ if and only if $m_1 \dots m_{i-1}$ contains $a$ and $o_1 \dots o_{j-1}$ contains $b$ (or the other way round).*

**Algorithm 3:** The algorithm computing $V_n$ for all nodes $n$

```
    Input:  a CFG $G = (V, E)$
    Output: $V_n = \{m \in V \mid m$ is on all maximal paths from $n\}$ for all $n \in V$

 1  Procedure VISIT($n, r$)                                  // Auxiliary procedure
 2      $n.counter \leftarrow n.counter - 1$
 3      if $n.counter = 0 \land r \notin V_n$ then
 4          $V_n \leftarrow V_n \cup \{r\}$
 5          for $m \in Predecessors(n)$ do
 6              VISIT($m, r$)
 7
 8  Procedure COMPUTE($n$)     // 'Coloring the graph red' for a given $n$
 9      for $m \in V$ do
10          $m.counter \leftarrow |Successors(m)|$
11      $V_n \leftarrow V_n \cup \{n\}$
12      for $m \in Predecessors(n)$ do
13          VISIT($m, n$)
14
15  Procedure COMPUTE$V_p$S    // Computation of sets $V_n$ for all nodes $n$
16      for $n \in V$ do
17          $V_n \leftarrow \emptyset$
18      for $n \in V$ do
19          COMPUTE($n$)
```

### 4.3 New Algorithm for DOD: Pseudocode and Complexity

Our DOD algorithm is shown in Algorithms 3 and 4. As nearly all applications of DOD need also NTSCD, we present the algorithm with a simple extension (gray lines with asterisks) that simultaneously computes NTSCD.

The DOD algorithm starts at line 20 of Algorithm 4. The first step is to compute the sets $V_p$ for all predicate nodes $p$ of a given CFG $G$. The computation of predicate nodes can be found in Algorithm 3. It is a slightly modified version of Algorithm 2. Recall that the procedure COMPUTE($n$) of Algorithm 2 marks red every node such that all maximal paths from the node contain $n$. The procedure COMPUTE($n$) of Algorithm 3 does in principle the same, but instead of the red color it marks the nodes with the identifier of the node $n$. Every node $m$ collects these marks in set $V_m$. After we run COMPUTE($n$) for all the nodes $n$ in the graph, each node $m$ has in its set $V_m$ precisely all nodes that are on all maximal paths from $m$. For the computation of DOD, only the sets $V_p$ for predicate nodes $p$ are needed, but the extension computing NTSCD may use all these sets.

When the sets $V_p$ are calculated, we compute DOD (and NTSCD) dependencies for each predicate node separately by procedures COMPUTEDOD($p$) and COMPUTENTSCD($p$). The procedure COMPUTEDOD($p$) first constructs the graph $A_p$ with the use of BUILD$A_p$($p$). Nodes of the graph are these of $V_p$. To compute edges, we trigger depth-first search in $G$ from each $n \in V_p$. If we find a node $m \in V_p$, we add the edge $(n, m)$ to the graph $A_p$ and stop the search on

**Algorithm 4:** The new DOD algorithm which computes also NTSCD if the gray lines are included (COMPUTE$V_p$S is given in Algorithm 3)

**Input:** a CFG $G = (V, E)$

**Output:** the DOD relation stored in *dod* (and NTSCD stored in *ntscd*)

```
1    Procedure COMPUTEDOD($p$) // Computation of DOD for predicate $p$
2        $A_p \leftarrow$ BUILD$A_p(p)$    // Get the graph $A_p$
3        if $A_p$ does not have the right shape then
4            return $\emptyset$
5        $S_1, S_2 \leftarrow$ COMPUTE$S_1S_2(p)$    // Get the sets $S_1, S_2$
6        if $S_1 \cap S_2 \neq \emptyset$ then
7            return $\emptyset$
8        $n_1n_2 \dots n_t \leftarrow$ UNFOLDCYCLE$(A_p, S_1)$    // Unfold the cycle of $A_p$
9        $U \leftarrow V_p \smallsetminus (S_1 \cup S_2)$
10       if $n_1n_2 \dots n_t \notin (S_1.U^*)^+.(S_2.U^*)^+.(S_1.U^*)^*$ then    // Apply Thm. 3
11           return $\emptyset$
12       $m_1 \dots m_i \leftarrow$ EXTRACT$(n_1n_2 \dots n_t, S_1.U^*.S_2)$    // Apply Thm. 4
13       $o_1 \dots o_j \leftarrow$ EXTRACT$(n_1n_2 \dots n_t, S_2.U^*.S_1)$
14       return $\{p \xrightarrow{\text{DOD}} \{a, b\} \mid a \in \{m_1, \dots, m_{i-1}\}, b \in \{o_1, \dots, o_{j-1}\}\}$
15
*16  Procedure COMPUTENTSCD($p$)    // Computation of NTSCD for predicate $p$
*17      $\{s_1, s_2\} \leftarrow Successors(p)$
*18      return $\{p \xrightarrow{\text{NTSCD}} n \mid n \in (V_{s_1} \smallsetminus V_{s_2}) \cup (V_{s_2} \smallsetminus V_{s_1})\}$
19
20   COMPUTE$V_p$S    // Computation of DOD and NTSCD for all nodes
21   $dod \leftarrow \emptyset$
*22  $ntscd \leftarrow \emptyset$
23   for $p \in Predicates(G)$ do
24       $dod \leftarrow dod \cup$ COMPUTEDOD$(p)$
*25      $ntscd \leftarrow ntscd \cup$ COMPUTENTSCD$(p)$
```

this path. When the graph $A_p$ is constructed, we check whether it has the right shape. If not, we return $\emptyset$ as there are no nodes DOD on $p$ in this case.

The next step is to compute the sets $S_1$ and $S_2$. Again, we apply a similar depth-first search as in the construction of $A_p$ described above. If the sets $S_1, S_2$ are not disjoint, we return $\emptyset$ as there are no nodes DOD on $p$.

Then we unfold the cycle in $A_p$ from an arbitrary node in $S_1$, compute the set $U$, and check whether the unfolding matches $(S_1.U^*)^+.(S_2.U^*)^+.(S_1.U^*)^*$. Note that any unfolding starting in $S_1$ matches this language iff the cycle has an unfolding of the form $S_1.U^*.(S_2.U^*)^*.S_2.U^*.(S_1.U^*)^*$ of Theorem 3. Hence, we return $\emptyset$ if the check fails.

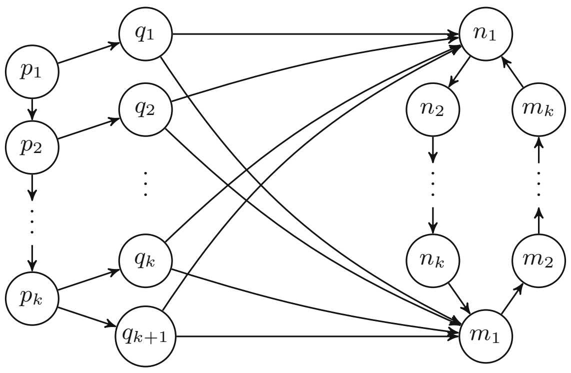


**Fig. 7.** A CFG with $|V|$ nodes that has the DOD relation of size $\Theta(|V|^3)$.

Finally, we extract the paths of the form $S_1.U^*.S_2$ and $S_2.U^*.S_1$ from the unfolding. Note that the last node of the latter path can be the first node of the unfolding. Finally, we compute the DOD dependencies according to Theorem 4.

The procedure COMPUTENTSCD($p$) used for the computation of NTSCD simply follows Definition 2: it makes dependent on $p$ each node that is on all maximal paths from the successor $s_1$ but not on all maximal paths from the successor $s_2$ or symmetrically for $s_2$ and $s_1$.

As the correctness of our algorithm comes directly from the observations made in the previous subsection, it remains only to analyze its complexity. The procedure COMPUTE$V_p$S consists of two cycles in sequence. The first cycle runs in $O(|V|)$. The second cycle calls $O(|V|)$-times the procedure COMPUTE($n$). This procedure is essentially identical to the procedure of the same name in Algorithm 2 and so is its time complexity, namely $O(|V| + |E|)$. Note that sets can be represented by bitvectors and therefore adding an element and checking the presence of an element in a set are constant-time. Overall, the procedure COMPUTE$V_p$S runs in $O(|V| \cdot (|V| + |E|))$, which is $O(|V|^2)$ for CFGs.

Now we discuss the complexity of the procedure COMPUTEDOD($p$). Creating the graph $A_p$ requires calling depth-first search $O(|V|)$ times, which yields $O(|V| \cdot |E|)$ in total. Computation of $S_1, S_2$ requires another two calls of depth-first search, which is in $O(|E|)$. When sets are represented as bitvectors, checking that $S_1$ and $S_2$ are disjoint is in $O(|V|)$. Unfolding the cycle, matching the unfolding to the language (line 10), and the procedure EXTRACT run also in $O(|V|)$. The construction of the DOD relation on line 14 is in $O(|V|^2)$. Altogether, COMPUTEDOD($p$) runs in $O(|V| \cdot |E| + |V|^2)$ which simplifies to $O(|V|^2)$ for CFGs.

COMPUTEDOD is called $O(|V|)$ times, so the overall complexity of computing DOD for a CFG $G = (V, E)$ is $O(|V|^3)$. If we compute also NTSCD, we make $O(|V|)$ extra calls to COMPUTENTSCD($p$), where one call takes $O(|V|)$ time. Therefore, the asymptotic complexity of computing NTSCD with DOD does not change from computing DOD only.

Our algorithm running in time $O(|V|^3)$ is asymptotically optimal as there exist graphs with DOD relations of size $\Theta(|V|^3)$. For example, the CFG in Fig. 7 has $|V| = 4k + 1$ nodes and the corresponding DOD relation

$$\{q_i \xrightarrow{\mathsf{DOD}} \{n_j, m_l\} \mid i \in \{1, \dots, k+1\}, j, l \in \{1, \dots, k\}\}$$

is of size $k^3 + k^2 \in \Theta(|V|^3)$.

## 5 Comparison to Control Closures

In 2011, Danicic et al. [13] introduced *control closures (CC)* that generalize control dependence from CFGs to arbitrary graphs. In particular, *strong control closure*, which is sensitive to non-termination, generalizes strong control dependence including NTSCD and DOD.

**Definition 8 (Strongly control-closed set).** *Let $G = (V, E)$ be a CFG and let $U \subseteq V$. The set $U$ is* strongly control-closed[3] *in $G$ if and only if for every node $v \in V \smallsetminus U$ that is reachable in $G$ from a node in $U$, one of these holds:*

- *there is no node in $U$ reachable from $v$ or*
- *there exists a node $u \in U$ such that all maximal paths from $v$ contain $u$ and it is the first node from $U$ on all these paths.*

In other words, whenever we leave a strongly control-closed set, we either cannot return back or we have to return back to the set in a certain node.

**Definition 9 (Strong control closure, strong CC).** *Let $G = (V, E)$ be a CFG and $V' \subseteq V$. A* strong control closure (strong CC) *of $V'$ is a strongly control-closed set $U \supseteq V'$ such that there is no strongly control-closed set $U'$ satisfying $U \supsetneq U' \supseteq V'$.*

Danicic et al. present an algorithm for the computation of strong control closures running in $O(|V|^4)$ [13, Theorem 66]. In fact, the algorithm uses a procedure $\Gamma$ that is very similar to our procedure COMPUTE($n$) of Algorithm 2.

We can also define the closure of a set of nodes under NTSCD and DOD.

**Definition 10 (NTSCD and DOD closure).** *Let $G = (V, E)$ be a CFG. A* NTSCD and DOD closure *of a set $V' \subseteq V$ is the smallest set $U \supseteq V'$ satisfying*

$$(n \in U \land p \xrightarrow{\mathsf{NTSCD}} n) \implies p \in U \quad \text{and} \quad (a, b \in U \land p \xrightarrow{\mathsf{DOD}} \{a, b\}) \implies p \in U.$$

Definition 10 directly provides an algorithm computing the NTSCD and DOD closure of a given set $V' \subseteq V$. Roughly speaking, if we represent the NTSCD relation with edges and the DOD relation with hyperedges in a directed hypergraph with nodes $V$, the closure computation amounts to gathering backward reachable nodes from $V'$.

[3] We adjusted the definition to the fact that predicates in our CFGs always have two outgoing edges (i.e., they are *complete* in terms of Danicic et al. [13]). The original definition [13] works with CFGs where each predicate has at most two successors and considers also paths that may end in a predicate with less than two successors.

Danicic et al. [13, Lemmas 93 and 94] proved that for a CFG $G = (V, E)$ with a distinguished *start* node from which all nodes in $V$ are reachable and a subset $U \subseteq V$ such that $start \in U$, the set $U$ is strongly control-closed iff it is closed under NTSCD and DOD. Hence, on graphs with such a *start* node, the strong CC of a set $V'$ containing the *start* node can be computed also by computing its NTSCD and DOD closure. Computation of the NTSCD and DOD closure runs in $O(|V|^3)$ as the backward reachability is dominated by the computation of NTSCD and DOD relations.

A substantial difference between the algorithm for strong CC by Danicic et al. [13] and our algorithm is that we are able to compute DOD and NTSCD separately, whereas the former is not. Moreover, our algorithm for NTSCD and DOD closure is asymptotically faster.

## 6 Experimental Evaluation

We implemented our algorithms for the computation of NTSCD, DOD, and the NTSCD and DOD closure in C++ on top of the LLVM [25] infrastructure. The implementation is a part of the library for program analysis and slicing called DG [6], which is used for example in the verification and test generation tool Symbiotic [7]. We also implemented the original Ranganath et al.'s algorithms for NTSCD and DOD, the fixed versions of these algorithms from Subsects. 3.1 and 4.1, and the algorithm for the computation of strong CC by Danicic et al.

In the implementation of the strong CC algorithm by Danicic et al. [13], we use our procedure COMPUTE($n$) of Algorithm 2 to implement the function $\Gamma$. This should have only a positive effect as this procedure is more efficient than iterating over all edges in a copy of the graph and removing them [13].

In our experiments, we use CFGs of functions (where nodes of the CFG represent basic blocks of the function) obtained in the following way. We took all benchmarks from the *Competition on Software Verification (SV-COMP)* 2020.[4] These benchmarks contain many artificial or generated code, but also a lot of real-life code, e.g., from the Linux project. Each source code file was compiled with CLANG into LLVM and preprocessed by the `-lowerswitch` pass to ensure that every basic block has at most two successors. Then we extracted individual functions and removed those with less than 100 basic blocks, as the computation of control dependence runs swiftly on small graphs. Because it is possible that one function is present in multiple benchmarks, the next step was to remove these duplicate functions. For every function, we computed the number of nodes and edges in its CFG, and performed DFS on the CFG to obtain the number of tree, forward, cross and back edges, and the depth of the DFS tree. If two or more functions shared the name and all the computed numbers, we kept only one such function. Note that this process may have removed also a function that was not a duplicate of some other, but only with a low probability. At the end, we were left with 2440 functions. The biggest function has 27851 basic blocks. Table 2 shows the distribution of the sizes of the generated CFGs.

[4] https://github.com/sosy-lab/sv-benchmarks, tag `svcomp20`.

**Table 2.** The *numbers* of considered CFGs by their *sizes*. The size of a CFG is the number of its nodes, which is the number of basic blocks of the corresponding function.

| size | number | size | number | size | number |
|---|---|---|---|---|---|
| 100 – 199 | 1713 | 500 – 599 | 35 | 900 – 999 | 3 |
| 200 – 299 | 355 | 600 – 699 | 29 | 1000 – 1999 | 23 |
| 300 – 399 | 159 | 700 – 799 | 18 | 2000 – 9999 | 22 |
| 400 – 499 | 73 | 800 – 899 | 7 | ≥ 10000 | 3 |

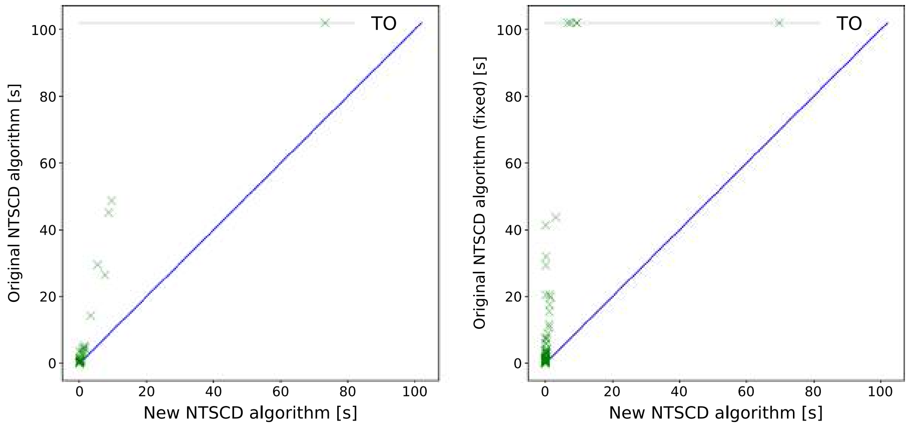


**Fig. 8.** Comparison of the running times of the new NTSCD algorithm and the incorrect (left) and the fixed (right) versions of the original NTSCD algorithm. TO stands for timeout.

The experiments were run on machines with *AMD EPYC* CPU with the frequency 3.1 GHz. Each benchmark run was constrained to 1 core and 8 GB of RAM. We used the tool *Benchexec* [4] to enforce resources isolation and to measure their usage. All presented times are CPU times. We set the timeout to 100 s for each algorithm run.

In the following, *original* algorithms refers to the algorithms of Ranganath et al. (we distinguish between the incorrect and the fixed versions when needed) and *new* algorithms refers to the algorithms introduced in this paper.

**NTSCD Algorithms.** In the first set of experiments, we compared the new algorithm for NTSCD against the incorrect and the fixed version of the original NTSCD algorithm. Although it seems that comparing to the incorrect version is meaningless, we did not want to compare only to the fixed version as the provided fix slows down the algorithm.

The results are depicted in Fig. 8. On the left scatter plot, there is the comparison of the new algorithm to the incorrect original algorithm and on the right scatter plot we compare to the fixed original algorithm. As we can see,

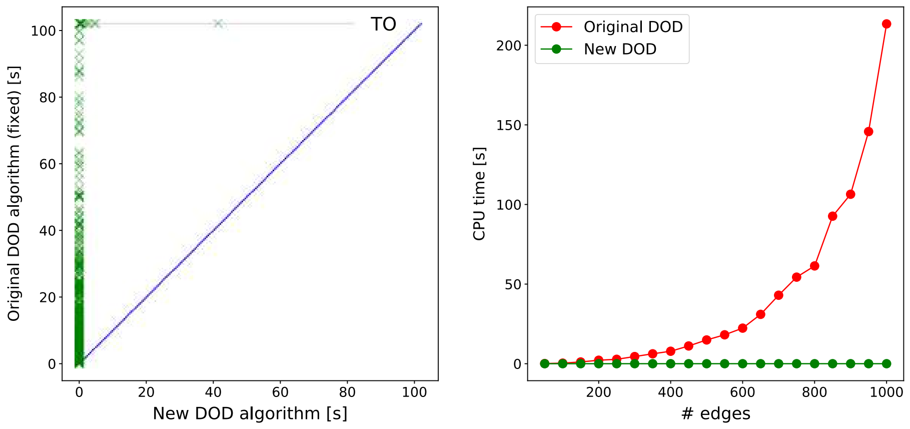


**Fig. 9.** Comparison of the running times of the new and the (fixed) original DOD algorithm. We use the considered benchmarks (left) and random graphs with 500 nodes and the number of edges specified by the $x$-axis (right).

the new algorithm outperforms the original algorithm significantly. The incorrect original algorithm produced a wrong NTSCD relation in 98.6 % of the considered benchmarks. The fixed version of the original algorithm returned precisely the same NTSCD relations as the new algorithm. We can also see that the scatter plot on the right contains more timeouts of the original algorithm. It supports the claim that the fix slows down the original algorithm.

**DOD Algorithms.** We compared the new DOD algorithm to the fixed version of the original DOD algorithm. As the fix does not change the asymptotic complexity of the original algorithm, we do not compare the new algorithm with the incorrect version of the original algorithm. The results of the experiments are displayed in Fig. 9 (left). We can see that the new algorithm is again very fast. In fact, the results resemble the results of the pure NTSCD algorithm, which is basically the part of the DOD algorithm that computes $V_p$ sets. It benefits from early checks that detect predicate nodes with no DOD dependencies.

As mentioned in the introduction, DOD is empty for structured programs as their CFGs are reducible. We do not know precisely how many of the 2440 considered functions have irreducible CFGs, but we know that 2373 of them use **goto** statements. DOD relations for 12 functions was non-empty, which means that CFGs of these functions are irreducible. Note that there may have been other irreducible CFGs with empty DOD relation.

Additionally, we tested the DOD algorithms on randomly generated graphs, where we can expect that irreducible graphs emerge more often. Figure 9 (right) shows the results for graphs that have 500 nodes and 50, 100, 150, . . . randomly distributed edges (such that every node has at most two successors). Each presented running time is in fact an average of 10 measurements with different random graphs. We can see that the new algorithm is agnostic to the number of

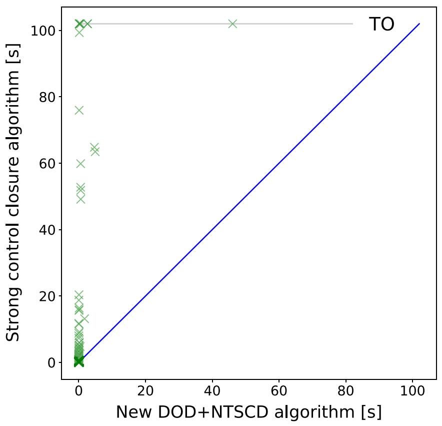


**Fig. 10.** Comparison of the running times of the strong CC algorithm by Danicic et al. [13] and our algorithm for the NTSCD and DOD closure.

edges. Its running time in this experiment ranges from $4.12 \cdot 10^{-3}$ to $8.89 \cdot 10^{-3}$ seconds. The original DOD algorithm does not scale well with the increasing number of edges.

**Strong CC Algorithm.** We also compare the strong CC algorithm of Danicic et al. [13] against our NTSCD and DOD closure algorithm on sets of nodes containing a distinguished *start* node, where these two algorithms produce equivalent results. For these experiments, we need a starting set that is going to be closed. We decided to run these experiments on the considered functions that have at least two exit points. The starting set consists of the node representing the entry point and the node representing one of the exit points. The closure of this set contains all nodes that may influence getting to the other exit points. The results are shown on the scatter plot in Fig. 10. Our algorithm clearly outperforms the strong CC algorithm.

## 7 Conclusion

We studied algorithms for the computation of strong control dependence, namely non-termination sensitive control dependence (NTSCD) and decisive order dependence (DOD) by Ranganath et al. [33] and strong control closures (strong CC) by Danicic et al. [13] on control flow graphs where each branching statement has two successors. We have demonstrated flaws in the original algorithms for computation of NTSCD and DOD and we have suggested corrections. Moreover, we have introduced new algorithms for NTSCD, DOD, and strong CC that are asymptotically faster. All the mentioned algorithms have been implemented and our experiments confirm dramatically better performance of the new algorithms.

## References

1. Amtoft, T.: Slicing for modern program structures: a theory for eliminating irrelevant loops. Inf. Process. Lett. **106**(2), 45–51 (2008). https://doi.org/10.1016/j.ipl.2007.10.002
2. Androutsopoulos, K., Clark, D., Harman, M., Krinke, J., Tratt, L.: State-based model slicing: a survey. ACM Comput. Surv. **45**(4), 53:1-53:36 (2013). https://doi.org/10.1145/2501654.2501667
3. Androutsopoulos, K., Clark, D., Harman, M., Li, Z., Tratt, L.: Control dependence for extended finite state machines. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 216–230. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_15
4. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
5. Bilardi, G., Pingali, K.: A framework for generalized control dependence. In: PLDI 1996, pp. 291–300. ACM (1996). https://doi.org/10.1145/231379.231435
6. Chalupa, M.: DG: analysis and slicing of LLVM bitcode. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 557–563. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_33
7. Chalupa, M., Jašek, T., Novák, J., Řechtáčková, A., Šoková, V., Strejček, J.: Symbiotic 8: beyond symbolic execution. In: TACAS 2021. LNCS, vol. 12652, pp. 453–457. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72013-1_31
8. Chalupa, M., Klaška, D., Strejček, J., Tomovič, L.: Fast computation of strong control dependencies. CoRR abs/2011.01564 (2020). https://arxiv.org/abs/2011.01564
9. Chalupa, M., Strejček, J.: Evaluation of program slicing in software verification. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) IFM 2019. LNCS, vol. 11918, pp. 101–119. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34968-4_6
10. Chen, F., Roşu, G.: Parametric and termination-sensitive control dependence. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 387–404. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_25
11. Cooper, K., Harvey, T., Kennedy, K.: Iterative data- ow analysis, revisited (2002)
12. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991). https://doi.org/10.1145/115372.115320
13. Danicic, S., Barraclough, R.W., Harman, M., Howroyd, J., Kiss, Á., Laurence, M.R.: A unifying theory of control dependence and its application to arbitrary program structures. Theor. Comput. Sci. **412**(49), 6809–6842 (2011). https://doi.org/10.1016/j.tcs.2011.08.033
14. Darte, A., Silber, G.-A.: Temporary arrays for distribution of loops with control dependences. In: Bode, A., Ludwig, T., Karl, W., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 357–367. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44520-X_47
15. Denning, D.E., Denning, P.J.: Certification of programs for secure information ow. Commun. ACM **20**(7), 504–513 (1977). https://doi.org/10.1145/359636.359712
16. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987). https://doi.org/10.1145/24039.24041

17. Gallagher, K., Binkley, D.: Program slicing. In: FoSM 2008, pp. 58–67 (2008). https://doi.org/10.1109/FOSM.2008.4659249
18. Giffhorn, D.: Slicing of concurrent programs and its application to information flow control. Ph.D. thesis, Karlsruhe Institute of Technology (2012). http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028814
19. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information ow control based on program dependence graphs. Int. J. Inf. Sec. **8**(6), 399–422 (2009). https://doi.org/10.1007/s10207-009-0086-1
20. Harrold, M.J., Rothermel, G., Sinha, S.: Computation of interprocedural control dependence. In: ISSTA 1998, pp. 11–20. ACM (1998). https://doi.org/10.1145/271771.271780
21. Hecht, M.S., Ullman, J.D.: Characterizations of reducible ow graphs. J. ACM **21**(3), 367–375 (1974). https://doi.org/10.1145/321832.321835
22. Horwitz, S., Reps, T.W., Binkley, D.W.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990). https://doi.org/10.1145/77606.77608
23. Khanfar, H., Lisper, B., Masud, A.N.: Static backward program slicing for safety-critical systems. In: de la Puente, J.A., Vardanega, T. (eds.) Ada-Europe 2015. LNCS, vol. 9111, pp. 50–65. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19584-1_4
24. Labbé, S., Gallois, J.: Slicing communicating automata specifications: polynomial algorithms for model reduction. Formal Aspects Comput. **20**(6), 563–595 (2008). https://doi.org/10.1007/s00165-008-0086-3
25. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: CGO 2004, pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
26. Léchenet, J., Kosmatov, N., Gall, P.L.: Cut branches before looking for bugs: certifiably sound verification on relaxed slices. Formal Asp. Comput. **30**(1), 107–131 (2018). https://doi.org/10.1007/s00165-017-0439-x
27. Loyall, J.P., Mathisen, S.A.: Using dependence analysis to support the software maintenance process. In: ICSM 1993, pp. 282–291. IEEE Computer Society (1993). https://doi.org/10.1109/ICSM.1993.366934
28. Lucia, A.D.: Program slicing: methods and applications. In: SCAM 2001, pp. 144–151. IEEE Computer Society (2001). https://doi.org/10.1109/SCAM.2001.972675
29. Metta, R., Becker, M., Bokil, P., Chakraborty, S., Venkatesh, R.: TIC: a scalable model checking based approach to WCET estimation. In: LCTES 2016, pp. 72–81. ACM (2016). https://doi.org/10.1145/2907950.2907961
30. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: FSE 1984, pp. 177–184. ACM (1984). https://doi.org/10.1145/800020.808263
31. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Software Eng. **16**(9), 965–979 (1990). https://doi.org/10.1109/32.58784
32. Ranganath, V.P., Amtoft, T., Banerjee, A., Dwyer, M.B., Hatcliff, J.: A new foundation for control-dependence and slicing for modern program structures. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 77–93. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_7
33. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. **29**(5), 27 (2007). https://doi.org/10.1145/1275497.1275502

34. Sinha, S., Harrold, M.J., Rothermel, G.: Interprocedural control dependence. ACM Trans. Softw. Eng. Methodol. **10**(2), 209–254 (2001). https://doi.org/10.1145/367008.367022
35. Stanier, J., Watson, D.: A study of irreducibility in C programs. Softw. Pract. Exp. **42**(1), 117–130 (2012). https://doi.org/10.1002/spe.1059
36. Tšahhirov, I., Laud, P.: Application of dependency graphs to security protocol analysis. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 294–311. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78663-4_20
37. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984). https://doi.org/10.1109/TSE.1984.5010248

# DIFFY: Inductive Reasoning of Array Programs Using Difference Invariants

Supratik Chakraborty[1], Ashutosh Gupta[1], and Divyesh Unadkat[1,2](✉)

[1] Indian Institute of Technology Bombay, Mumbai, India
{supratik,akg}@cse.iitb.ac.in
[2] TCS Research, Pune, India
divyesh.unadkat@tcs.com

**Abstract.** We present a novel verification technique to prove properties of a class of array programs with a symbolic parameter $N$ denoting the size of arrays. The technique relies on constructing two slightly different versions of the same program. It infers difference relations between the corresponding variables at key control points of the joint control-flow graph of the two program versions. The desired post-condition is then proved by inducting on the program parameter $N$, wherein the difference invariants are crucially used in the inductive step. This contrasts with classical techniques that rely on finding potentially complex loop invaraints for each loop in the program. Our synergistic combination of inductive reasoning and finding simple difference invariants helps prove properties of programs that cannot be proved even by the winner of Arrays sub-category in SV-COMP 2021. We have implemented a prototype tool called DIFFY to demonstrate these ideas. We present results comparing the performance of DIFFY with that of state-of-the-art tools.

## 1 Introduction

Software used in a wide range of applications use arrays to store and update data, often using loops to read and write arrays. Verifying correctness properties of such array programs is important, yet challenging. A variety of techniques have been proposed in the literature to address this problem, including inference of quantified loop invariants [20]. However, it is often difficult to automatically infer such invariants, especially when programs have loops that are sequentially composed and/or nested within each other, and have complex control flows. This has spurred recent interest in mathematical induction-based techniques for verifying parametric properties of array manipulating programs [11,12,42,44]. While induction-based techniques are efficient and quite powerful, their Achilles heel is the automation of the inductive argument. Indeed, this often becomes the limiting step in applications of induction-based techniques. Automating the induction step and expanding the class of array manipulating programs to which induction-based techniques can be applied forms the primary motivation for our work. Rather than being a stand-alone technique, we envisage our work being used as part of a portfolio of techniques in a modern program verification tool.

We propose a novel and practically efficient induction-based technique that advances the state-of-the-art in automating the inductive step when reasoning about array manipulating programs. This allows us to automatically verify interesting properties of a large class of array manipulating programs that are beyond the reach of state-of-the-art induction-based techniques, viz. [12,42]. The work that comes closest to us is Vajra [12], which is part of the portfolio of techniques in VeriAbs [1] – the winner of SV-COMP 2021 in the Arrays Reach sub-category. Our work addresses several key limitations of the technique implemented in Vajra, thereby making it possible to analyze a much larger class of array manipulating programs than can be done by VeriAbs. Significantly, this includes programs with nested loops that have hitherto been beyond the reach of automated techniques that use mathematical induction [12,42,44].

A key innovation in our approach is the construction of two slightly different versions of a given program that have identical control flow structures but slightly different data operations. We automatically identify simple relations, called *difference invariants*, between corresponding variables in the two versions of a program at key control flow points. Interestingly, these relations often turn out to be significantly simpler than inductive invariants required to prove the property directly. This is not entirely surprising, since the difference invariants depend less on what individual statements in the programs are doing, and more on the difference between what they are doing in the two versions of the program. We show how the two versions of a given program can be automatically constructed, and how differences in individual statements can be analyzed to infer simple difference invariants. Finally, we show how these difference invariants can be used to simplify the reasoning in the inductive step of our technique.

We consider programs with (possibly nested) loops manipulating arrays, where the size of each array is a symbolic integer parameter $N$ $(> 0)$[1]. We verify (a sub-class of) quantified and quantifier-free properties that may depend on the symbolic parameter $N$. Like in [12], we view the verification problem as one of proving the validity of a parameterized Hoare triple $\{\varphi(N)\}$ $\mathsf{P}_N$ $\{\psi(N)\}$ for all values of $N$ $(> 0)$, where arrays are of size $N$ in the program $\mathsf{P}_N$, and $N$ is a free variable in $\varphi(\cdot)$ and $\psi(\cdot)$.

To illustrate the kind of programs that are amenable to our technique, consider the program shown in Fig. 1(a), adapted from an SV-COMP benchmark. This program has a couple of sequentially composed loops that update arrays and scalars. The scalars `S` and `F` are initialized to 0 and 1 respectively before the first loop starts iterating. Subsequently, the first loop computes a recurrence in variable `S` and initializes elements of the array `B` to 1 if the corresponding elements of array `A` have non-negative values, and to 0 otherwise. The outermost branch condition in the body of the second loop evaluates to true only if the program parameter $N$ and the variable `S` have same values. The value of `F` is reset based on some conditions depending on corresponding entries of arrays `A` and `B`. The pre-condition of this program is `true`; the post-condition asserts that `F` is never reset in the second loop.

[1] For a more general class of programs supported by our technique, please see [13].

```
// assume(true)
S = 0; F = 1;
for(i = 0; i< N; i++) {
 S = S + 1;
 if ( A[i] >= 0 ) B[i] = 1;
 else B[i] = 0;
}
for(j = 0; j< N; j++) {
 if(S == N) {
  if ( A[j] >= 0 && !B[j] ) F = 0;
  if ( A[j] < 0 && B[j] ) F = 0;
}
}
// assert(F == 1)
```

(a)

```
// assume(true)
S = 0;
for(i=0; i<N; i++) A[i] = 0;
for(j=0; j<N; j++) S = S + 1;
for(k=0; k<N; k++) {
 for(l=0; l<N; l++) A[l] = A[l] + 1;
 A[k] = A[k] + S;
}
// assert(forall x in [0,N), A[x]==2*N)
```

(b)

**Fig. 1.** Motivating examples

State-of-the-art techniques find it difficult to prove the assertion in this program. Specifically, Vajra [12] is unable to prove the property, since it cannot reason about the branch condition (in the second loop) whose value depends on the program parameter $N$. VeriAbs [1], which employs a sequence of techniques such as loop shrinking, loop pruning, and inductive reasoning using [12] is also unable to verify the assertion shown in this program. Indeed, the loops in this program cannot be merged as the final value of `S` computed by the first loop is required in the second loop; hence loop shrinking does not help. Also, loop pruning does not work due to the complex dependencies in the program and the fact that the exact value of the recurrence variable `S` is required to verify the program. Subsequent abstractions and techniques applied by VeriAbs from its portfolio are also unable to verify the given post-condition. VIAP [42] translates the program to a quantified first-order logic formula in the theory of equality and uninterpreted functions [32]. It applies a sequence of tactics to simplify and prove the generated formula. These tactics include computing closed forms of recurrences, induction over array indices and the like to prove the property. However, its sequence of tactics is unable to verify this example within our time limit of 1 min.

Benchmarks with nested loops are a long standing challenge for most verifiers. Consider the program shown in Fig. 1(b) with a nested loop in addition to sequentially composed loops. The first loop initializes entries in array `A` to 0. The second loop aggregates a constant value in the scalar `S`. The third loop is a nested loop that updates array `A` based on the value of `S`. The entries of `A` are updated in the inner as well as outer loop. The property asserts that on termination, each array element equals twice the value of the parameter $N$.

While the inductive reasoning of Vajra and the tactics in VIAP do not support nested loops, the sequence of techniques used by VeriAbs is also unable to prove the given post-condition in this program. In sharp contrast, our prototype tool Diffy is able to verify the assertions in both these programs automatically within a few seconds. This illustrates the power of the inductive technique proposed in this paper.

The technical contributions of the paper can be summarized as follows:

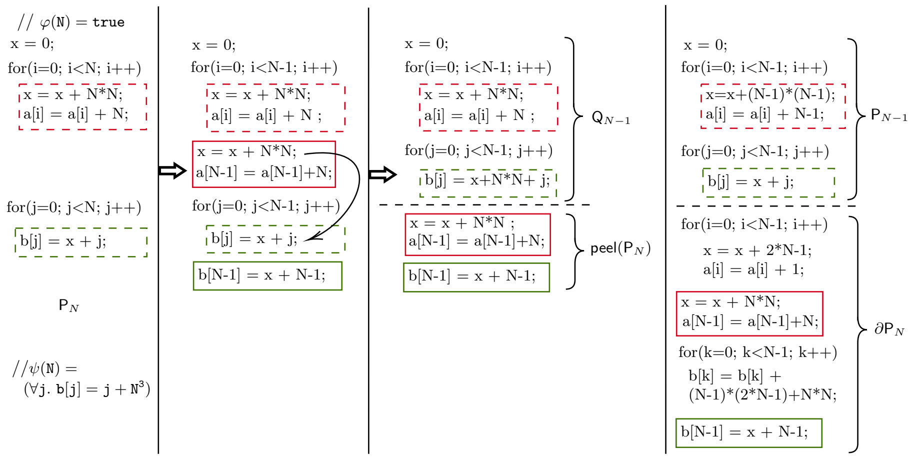
– We present a novel technique based on mathematical induction to prove interesting properties of a class of programs that manipulate arrays. The crucial inductive step in our technique uses difference invariants from two slightly different versions of the same program, and differs significantly from other induction-based techniques proposed in the literature [11,12,42,44].
– We describe algorithms to transform the input program for use in our inductive verification technique. We also present techniques to infer simple difference invariants from the two slightly different program versions, and to complete the inductive step using these difference invariants.
– We describe a prototype tool Diffy that implements our algorithms.
– We compare Diffy vis-a-vis state-of-the-art tools for verification of C programs that manipulate arrays on a large set of benchmarks. We demonstrate that Diffy significantly outperforms the winners of SV-COMP 2019, 2020 and 2021 in the Array Reach sub-category.

## 2 Overview and Relation to Earlier Work

In this section, we provide an overview of the main ideas underlying our technique. We also highlight how our technique differs from [12], which comes closest to our work. To keep the exposition simple, we consider the program $\mathsf{P}_N$, shown in the first column of Fig. 2, where $N$ is a symbolic parameter denoting the sizes of arrays `a` and `b`. We assume that we are given a parameterized pre-condition $\varphi(N)$, and our goal is to establish the parameterized post-condition $\psi(N)$, for all $N > 0$. In [12,44], techniques based on mathematical induction (on $N$) were proposed to solve this class of problems. As with any induction-based technique, these approaches consist of three steps. First, they check if the *base case* holds, i.e. if the Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ holds for small values of $N$, say $1 \leq N \leq M$, for some $M > 0$. Next, they assume that the *inductive hypothesis* $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds for some $N \geq M+1$. Finally, in the *inductive step*, they show that if the inductive hypothesis holds, so does $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$. It is not hard to see that the inductive step is the most crucial step in this style of reasoning. It is also often the limiting step, since not all programs and properties allow for efficient inferencing of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ from $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$.

Like in [12,44], our technique uses induction on $N$ to prove the Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ for all $N > 0$. Hence, our base case and inductive hypothesis are the same as those in [12,44]. However, our reasoning in the crucial inductive step is significantly different from that in [12,44], and this is where our primary contribution lies. As we show later, not only does this allow a much larger class of programs to be efficiently verified compared to [12,44], it also permits reasoning about classes of programs with nested loops, that are beyond the reach of [12,44]. Since the work of [12] significantly generalizes that of [44], henceforth, we only refer to [12] when talking of earlier work that uses induction on $N$.

In order to better understand our contribution and its difference vis-a-vis the work of [12], a quick recap of the inductive step used in [12] is essential. The


```
// φ(N) = true
x = 0;
for(i=0; i<N; i++)
  x = x + N*N;
  a[i] = a[i] + N;

for(j=0; j<N; j++)
  b[j] = x + j;

//ψ(N) =
  (∀j. b[j] = j + N^3)
```

```
x = 0;
for(i=0; i<N-1; i++)
  x = x + N*N;
  a[i] = a[i] + N ;
x = x + N*N;
a[N-1] = a[N-1]+N;
for(j=0; j<N-1; j++)
  b[j] = x + j;
b[N-1] = x + N-1;
```

```
x = 0;
for(i=0; i<N-1; i++)
  x = x + N*N;
  a[i] = a[i] + N ;
for(j=0; j<N-1; j++)
  b[j] = x+N*N+ j;
- - - - - - - - - - - - -
x = x + N*N ;
a[N-1] = a[N-1]+N;
b[N-1] = x + N-1;
```

```
x = 0;
for(i=0; i<N-1; i++)
  x=x+(N-1)*(N-1);
  a[i] = a[i] + N-1;
for(j=0; j<N-1; j++)
  b[j] = x + j;
- - - - - - - - - - - - -
for(i=0; i<N-1; i++)
  x = x + 2*N-1;
  a[i] = a[i] + 1;
x = x + N*N;
a[N-1] = a[N-1]+N;
for(k=0; k<N-1; k++)
  b[k] = b[k] +
  (N-1)*(2*N-1)+N*N;
b[N-1] = x + N-1;
```

**Fig. 2.** Pictorial depiction of our program transformations

inductive step in [12] crucially relies on finding a "difference program" $\partial\mathsf{P}_N$ and a "difference pre-condition" $\partial\varphi(N)$ such that: (i) $\mathsf{P}_N$ is semantically equivalent to $\mathsf{P}_{N-1};\partial\mathsf{P}_N$, where ';' denotes sequential composition of programs[2], (ii) $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$, and (iii) no variable/array element in $\partial\varphi(N)$ is modified by $\mathsf{P}_{N-1}$. As shown in [12], once $\partial\mathsf{P}_N$ and $\partial\varphi(N)$ satisfying these conditions are obtained, the problem of proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ can be reduced to that of proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$. This approach can be very effective if (i) $\partial\mathsf{P}_N$ is "simpler" (e.g. has fewer loops or strictly less deeply nested loops) than $\mathsf{P}_N$ and can be computed efficiently, and (ii) a formula $\partial\varphi(N)$ satisfying the conditions mentioned above exists and can be computed efficiently.

The requirement of $\mathsf{P}_N$ being semantically equivalent to $\mathsf{P}_{N-1};\partial\mathsf{P}_N$ is a very stringent one, and finding such a program $\partial\mathsf{P}_N$ is non-trivial in general. In fact, the authors of [12] simply provide a set of syntax-guided conditionally sound heuristics for computing $\partial\mathsf{P}_N$. Unfortunately, when these conditions are violated (we have found many simple programs where they are violated), there are no known algorithmic techniques to generate $\partial\mathsf{P}_N$ in a sound manner. Even if a program $\partial\mathsf{P}_N$ were to be found in an ad-hoc manner, it may be as "complex" as $\mathsf{P}_N$ itself. This makes the approach of [12] ineffective for analyzing such programs. As an example, the fourth column of Fig. 2 shows $\mathsf{P}_{N-1}$ followed by one possible $\partial\mathsf{P}_N$ that ensures $\mathsf{P}_N$ (shown in the first column of the same figure) is semantically equivalent to $\mathsf{P}_{N-1};\partial\mathsf{P}_N$. Notice that $\partial\mathsf{P}_N$ in this example has two sequentially composed loops, just like $\mathsf{P}_N$ had. In addition, the assignment statement in the body of the second loop uses a more complex expression than that present in the corresponding loop of $\mathsf{P}_N$. Proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$

[2] Although the authors of [12] mention that it suffices to find a $\partial\mathsf{P}_N$ that satisfies $\{\varphi(N)\}\ \mathsf{P}_{N-1};\partial\mathsf{P}_N\ \{\psi(N)\}$, they do not discuss any technique that takes $\varphi(N)$ or $\psi(N)$ into account when generating $\partial\mathsf{P}_N$.

may therefore not be any simpler (perhaps even more difficult) than proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$.

In addition to the difficulty of computing $\partial\mathsf{P}_N$, it may be impossible to find a formula $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$, as required by [12]. This can happen even for fairly routine pre-conditions, such as $\varphi(N) \equiv \big(\bigwedge_{i=0}^{N-1} A[i] = N\big)$. Notice that there is no $\partial\varphi(N)$ that satisfies $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$ in this case. In such cases, the technique of [12] cannot be used at all, even if $\mathsf{P}_N$, $\varphi(N)$ and $\psi(N)$ are such that there exists a trivial proof of $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$.

The inductive step proposed in this paper largely mitigates the above problems, thereby making it possible to efficiently reason about a much larger class of programs than that possible using the technique of [12]. Our inductive step proceeds as follows. Given $\mathsf{P}_N$, we first algorithmically construct two programs $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$, such that $\mathsf{P}_N$ is semantically equivalent to $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$. Intuitively, $\mathsf{Q}_{N-1}$ is the same as $\mathsf{P}_N$, but with all loop bounds that depend on $N$ now modified to depend on $N-1$ instead. Note that this is different from $\mathsf{P}_{N-1}$, which is obtained by replacing *all uses* (not just in loop bounds) of $N$ in $\mathsf{P}_N$ by $N-1$. As we will see, this simple difference makes the generation of $\mathsf{peel}(\mathsf{P}_N)$ significantly simpler than generation of $\partial\mathsf{P}_N$, as in [12]. While generating $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(P_N)$ may sound similar to generating $\mathsf{P}_{N-1}$ and $\partial\mathsf{P}_N$ [12], there are fundamental differences between the two approaches. First, as noted above, $\mathsf{P}_{N-1}$ is semantically different from $\mathsf{Q}_{N-1}$. Similarly, $\mathsf{peel}(\mathsf{P}_N)$ is also semantically different from $\partial\mathsf{P}_N$. Second, we provide an algorithm for generating $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ that works for a significantly larger class of programs than that for which the technique of [12] works. Specifically, our algorithm works for all programs amenable to the technique of [12], and also for programs that violate the restrictions imposed by the grammar and conditional heuristics in [12]. For example, we can algorithmically generate $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ even for a class of programs with arbitrarily nested loops – a program feature explicitly disallowed by the grammar in [12]. Third, we guarantee that $\mathsf{peel}(\mathsf{P}_N)$ is "simpler" than $\mathsf{P}_N$ in the sense that the maximum nesting depth of loops in $\mathsf{peel}(\mathsf{P}_N)$ is *strictly less* than that in $\mathsf{P}_N$. Thus, if $\mathsf{P}_N$ has no nested loops (all programs amenable to analysis by [12] belong to this class), $\mathsf{peel}(\mathsf{P}_N)$ is guaranteed to be loop-free. As demonstrated by the fourth column of Fig. 2, no such guarantees can be given for $\partial\mathsf{P}_N$ generated by the technique of [12]. This is a significant difference, since it greatly simplifies the analysis of $\mathsf{peel}(\mathsf{P}_N)$ vis-a-vis that of $\partial\mathsf{P}_N$.

We had mentioned earlier that some pre-conditions $\varphi(N)$ do not admit any $\partial\varphi(N)$ such that $\varphi(N) \Rightarrow \varphi(N-1) \wedge \partial\varphi(N)$. It is, however, often easy to compute formulas $\varphi'(N-1)$ and $\Delta\varphi'(N)$ in such cases such that $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$, and the variables/array elements in $\Delta\varphi'(N)$ are not modified by either $\mathsf{P}_{N-1}$ or $\mathsf{Q}_{N-1}$. For example, if we were to consider a (new) pre-condition $\varphi(N) \equiv \big(\bigwedge_{i=0}^{N-1} A[i] = N\big)$ for the program $\mathsf{P}_N$ shown in the first column of Fig. 2, then we have $\varphi'(N-1) \equiv \big(\bigwedge_{i=0}^{N-2} A[i] = N\big)$ and $\Delta\varphi'(N) \equiv \big(A[N-1] = N\big)$. We assume the availability of such a $\varphi'(N-1)$ and $\Delta\varphi'(N)$ for the given $\varphi(N)$. This significantly relaxes the requirement on pre-conditions and allows a much larger class of Hoare triples to be proved using our technique vis-a-vis that of [12].

The third column of Fig. 2 shows $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ generated by our algorithm for the program $\mathsf{P}_N$ in the first column of the figure. It is illustrative to compare these with $\mathsf{P}_{N-1}$ and $\partial\mathsf{P}_N$ shown in the fourth column of Fig. 2. Notice that $\mathsf{Q}_{N-1}$ has the same control flow structure as $\mathsf{P}_{N-1}$, but is not semantically equivalent to $\mathsf{P}_{N-1}$. In fact, $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$ may be viewed as closely related versions of the same program. Let $V_{\mathsf{Q}}$ and $V_{\mathsf{P}}$ denote the set of variables of $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$ respectively. We assume $V_{\mathsf{Q}}$ is disjoint from $V_{\mathsf{P}}$, and analyze the joint execution of $\mathsf{Q}_{N-1}$ starting from a state satisfying the pre-condition $\varphi'(N-1)$, and $\mathsf{P}_{N-1}$ starting from a state satisfying $\varphi(N-1)$. The purpose of this analysis is to compute a difference predicate $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ that relates corresponding variables in $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$ at the end of their joint execution. The above problem is reminiscent of (yet, different from) translation validation [4,17,24,40,46,48,49], and indeed, our calculation of $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is motivated by techniques from the translation validation literature. An important finding of our study is that corresponding variables in $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$ are often related by simple expressions on $N$, regardless of the complexity of $\mathsf{P}_N$, $\varphi(N)$ or $\psi(N)$. Indeed, in all our experiments, we didn't need to go beyond quadratic expressions on $N$ to compute $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$.

Once the steps described above are completed, we have $\Delta\varphi'(N)$, $\mathsf{peel}(\mathsf{P}_N)$ and $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$. It can now be shown that if the inductive hypothesis, i.e. $\{\varphi(N-1)\}\ \mathsf{P}_{N-1}\ \{\psi(N-1)\}$ holds, then proving $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ reduces to proving $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$, where $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$. A few points are worth emphasizing here. First, if $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is obtained as a set of equalities, the existential quantifier in the formula $\psi'(N-1)$ can often be eliminated simply by substitution. We can also use quantifier elimination capabilities of modern SMT solvers, viz. Z3 [39], to eliminate the quantifier, if needed. Second, recall that unlike $\partial\mathsf{P}_N$ generated by the technique of [12], $\mathsf{peel}(\mathsf{P}_N)$ is guaranteed to be "simpler" than $\mathsf{P}_N$, and is indeed loop-free if $\mathsf{P}_N$ has no nested loops. Therefore, proving $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$ is typically significantly simpler than proving $\{\psi(N-1) \wedge \partial\varphi(N)\}\ \partial\mathsf{P}_N\ \{\psi(N)\}$. Finally, it may happen that the pre-condition in $\{\Delta\varphi'(N) \wedge \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$ is not strong enough to yield a proof of the Hoare triple. In such cases, we need to strengthen the existing pre-condition by a formula, say $\xi'(N-1)$, such that the strengthened pre-condition implies the weakest pre-condition of $\psi(N)$ under $\mathsf{peel}(\mathsf{P}_N)$. Having a simple structure for $\mathsf{peel}(\mathsf{P}_N)$ (e.g., loop-free for the entire class of programs for which [12] works) makes it significantly easier to compute the weakest pre-condition. Note that $\xi'(N-1)$ is defined over the variables in $V_{\mathsf{Q}}$. In order to ensure that the inductive proof goes through, we need to strengthen the post-condition of the original program by $\xi(N)$ such that $\xi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1) \Rightarrow \xi'(N-1)$. Computing $\xi(N-1)$ requires a special form of logical abduction that ensures that $\xi(N-1)$ refers only to variables in $V_P$. However, if $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$ is given as a set of equalities (as is often the case), $\xi(N-1)$ can be computed from $\xi'(N-1)$ simply by substitution. This process of strengthening the pre-condition and post-condition may need to iterate a few times until a fixed point is reached, similar to what happens in the inductive step of [12]. Note that the fixed point iterations may not always converge (verification is undecidable in general). However, in our experiments, convergence always happened within a few iterations. If $\xi'(N-1)$ denotes the formula obtained on reaching the fixed point, the final Hoare triple to be proved is $\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{peel}(\mathsf{P}_N)$ $\{\xi(N) \wedge \psi(N)\}$, where $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$. Having a simple (often loop-free) $\mathsf{peel}(\mathsf{P}_N)$ significantly simplifies the above process.

We conclude this section by giving an overview of how $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ are computed for the program $\mathsf{P}_N$ shown in the first column of Fig. 2. The second column of this figure shows the program obtained from $\mathsf{P}_N$ by peeling the last iteration of each loop of the program. Clearly, the programs in the first and second columns are semantically equivalent. Since there are no nested loops in $\mathsf{P}_N$, the peels (shown in solid boxes) in the second column are loop-free program fragments. For each such peel, we identify variables/array elements modified in the peel and used in subsequent non-peeled parts of the program. For example, the variable `x` is modified in the peel of the first loop and used in the body of the second loop, as shown by the arrow in the second column of Fig. 2. We replace all such uses (if needed, transitively) by expressions on the right-hand side of assignments in the peel until no variable/array element modified in the peel is used in any subsequent non-peeled part of the program. Thus, the use of `x` in the body of the second loop is replaced by the expression `x + N * N` in the third column of Fig. 2. The peeled iteration of the first loop can now be moved to the end of the program, since the variables modified in this peel are no longer used in any subsequent non-peeled part of the program. Repeating the above steps for the peeled iteration of the second loop, we get the program shown in the third column of Fig. 2. This effectively gives a transformed program that can be divided into two parts: (i) a program $\mathsf{Q}_{N-1}$ that differs from $\mathsf{P}_N$ only in that all loops are truncated to iterate $N-1$ (instead of $N$) times, and (ii) a program $\mathsf{peel}(\mathsf{P}_N)$ that is obtained by concatenating the peels of loops in $\mathsf{P}_N$ in the same order in which the loops appeared in $\mathsf{P}_N$. It is not hard to see that $\mathsf{P}_N$, shown in the first column of Fig. 2, is semantically equivalent to $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$. Notice that the construction of $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ was fairly straightforward, and did not require any complex reasoning. In sharp contrast, construction of $\partial\mathsf{P}_N$, as shown in the bottom half of fourth column of Fig. 2, requires non-trivial reasoning, and produces a program with two sequentially composed loops.

## 3 Preliminaries and Notation

We consider programs generated by the grammar shown below:

$$\begin{array}{rl}
\mathsf{PB} ::= & \mathsf{St} \\
\mathsf{St} ::= & \mathsf{St}\ ;\ \mathsf{St} \mid\ v := \mathsf{E} \mid\ A[\mathsf{E}] := \mathsf{E} \mid\ \mathbf{if}(\mathsf{BoolE})\ \mathbf{then}\ \mathsf{St}\ \mathbf{else}\ \mathsf{St} \mid \\
 & \mathbf{for}\ (\ell := 0;\ \ell < \mathsf{UB};\ \ell := \ell{+}1)\ \ \{\mathsf{St}\} \\
\mathsf{E} ::= & \mathsf{E}\ \mathsf{op}\ \ \mathsf{E} \mid\ A[\mathsf{E}]\ \mid\ v\ \mid\ \ell\ \mid\ \mathsf{c}\ \mid\ N \\
\mathsf{op} ::= & +\ \mid\ \text{-}\ \mid\ *\ \mid\ / \\
\mathsf{UB} ::= & \mathsf{UB}\ \mathsf{op}\ \ \mathsf{UB} \mid\ \ell\ \mid\ \mathsf{c}\ \mid\ N \\
\mathsf{BoolE} ::= & \mathsf{E}\ \mathsf{relop}\ \mathsf{E} \mid\ \mathsf{BoolE}\ \mathsf{AND}\ \mathsf{BoolE}\ \mid\ \mathsf{NOT}\ \mathsf{BoolE}\ \mid\ \mathsf{BoolE}\ \mathsf{OR}\ \mathsf{BoolE}
\end{array}$$

Formally, we consider a program $\mathsf{P}_N$ to be a tuple $(\mathcal{V}, \mathcal{L}, \mathcal{A}, \mathsf{PB}, N)$, where $\mathcal{V}$ is a set of scalar variables, $\mathcal{L} \subseteq \mathcal{V}$ is a set of scalar loop counter variables, $\mathcal{A}$ is a set of array variables, $\mathsf{PB}$ is the program body, and $N$ is a special symbol denoting a positive integer parameter of the program. In the grammar shown above, we assume that $A \in \mathcal{A}$, $v \in \mathcal{V} \setminus \mathcal{L}$, $\ell \in \mathcal{L}$ and $\mathsf{c} \in \mathbb{Z}$. We also assume that each loop $\mathsf{L}$ has a unique loop counter variable $\ell$ that is initialized at the beginning of $\mathsf{L}$ and is incremented by 1 at the end of each iteration. We assume that the assignments in the body of $\mathsf{L}$ do not update $\ell$. For each loop $\mathsf{L}$ with termination condition $\ell < \mathsf{UB}$, we require that $\mathsf{UB}$ is an expression in terms of $N$, variables in $\mathcal{L}$ representing loop counters of loops that nest $\mathsf{L}$, and constants as shown in the grammar. Our grammar allows a large class of programs (with nested loops) to be analyzed using our technique, and that are beyond the reach of state-of-the-art tools like [1,12,42].

We verify Hoare triples of the form $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$, where the formulas $\varphi(N)$ and $\psi(N)$ are either universally quantified formulas of the form $\forall I\ (\alpha(I, N) \Rightarrow \beta(\mathcal{A}, \mathcal{V}, I, N))$ or quantifier-free formulas of the form $\eta(\mathcal{A}, \mathcal{V}, N)$. In these formulas, $I$ is a sequence of array index variables, $\alpha$ is a quantifier-free formula in the theory of arithmetic over integers, and $\beta$ and $\eta$ are quantifier-free formulas in the combined theory of arrays and arithmetic over integers.

For technical reasons, we rename all scalar and array variables in the program in a pre-processing step as follows. We rename each scalar variable using the well-known Static Single Assignment (SSA) [43] technique, such that the variable is written at (at most) one location in the program. We also rename arrays in the program such that each loop updates its own version of an array and multiple writes to an array element within the same loop are performed on different versions of that array. We use techniques for array SSA [30] renaming studied earlier in the context of compilers, for this purpose. In the subsequent exposition, we assume that scalar and array variables in the program are already SSA renamed, and that all array and scalar variables referred to in the pre- and post-conditions are also expressed in terms of SSA renamed arrays and scalars.

## 4 Verification Using Difference Invariants

The key steps in the application of our technique, as discussed in Sect. 2, are

A1: Generation of $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ from a given $\mathsf{P}_N$.
A2: Generation of $\varphi'(N-1)$ and $\Delta\varphi'(N)$ from a given $\varphi(N)$.
A3: Generation of the difference invariant $D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)$, given $\varphi(N-1)$, $\varphi'(N-1)$, $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$.
A4: Proving $\{\Delta\varphi'(N)\ \wedge\ \exists V_{\mathsf{P}}\big(\psi(N-1) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\psi(N)\}$, possibly by generation of $\xi'(N-1)$ and $\xi(N)$ to strengthen the pre- and post-conditions, respectively.

We now discuss techniques for solving each of these sub-problems.

## 4.1 Generating $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$

The procedure illustrated in Fig. 2 (going from the first column to the third column) is fairly straightforward if none of the loops have any nested loops within them. It is easy to extend this to arbitrary sequential compositions of non-nested loops. Having all variables and arrays in SSA-renamed forms makes it particularly easy to carry out the substitution exemplified by the arrow shown in the second column of Fig. 2. Hence, we don't discuss any further the generation of $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ when all loops are non-nested.

The case of nested loops is, however, challenging and requires additional discussion. Before we present an algorithm for handling this case, we discuss the intuition using an abstract example. Consider a pair of nested loops, $\mathsf{L}_1$ and $\mathsf{L}_2$, as shown in Fig. 3. Suppose that `B1` and `B3` are loop-free code fragments in the body of $\mathsf{L}_1$ that precede and succeed the nested loop $\mathsf{L}_2$. Suppose further that the loop body, `B2`, of $\mathsf{L}_2$ is loop-free. To focus on the key aspects of computing peels of nested loops, we make two simplifying assumptions: (i) no scalar variable or array element modified in `B2` is used subsequently (including transitively) in either `B3` or `B1`, and (ii) every scalar variable or array element that is modified in `B1` and used subsequently in `B2`, is not modified again in either `B1`, `B2` or `B3`. Note that these assumptions are made primarily to simplify the exposition. For a detailed discussion on how our technique can be used even with some relaxations of these assumptions, the reader is referred to [13]. The peel of the abstract loops $\mathsf{L}_1$ and $\mathsf{L}_2$ is as shown in Fig. 4. The first loop in the peel includes the last iteration of $\mathsf{L}_2$ in each of the $N-1$ iterations of $\mathsf{L}_1$, that was missed in $\mathsf{Q}_{N-1}$. The subsequent code includes the last iteration of $\mathsf{L}_1$ that was missed in $\mathsf{Q}_{N-1}$.

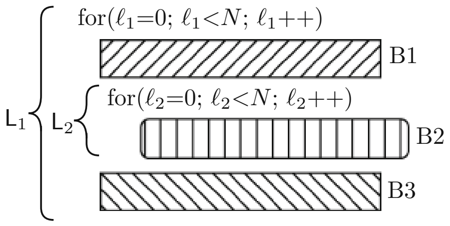

for($\ell_1$=0; $\ell_1$<$N$; $\ell_1$++)
B1
$\mathsf{L}_1$
$\mathsf{L}_2$
for($\ell_2$=0; $\ell_2$<$N$; $\ell_2$++)
B2
B3

**Fig. 3.** A generic nested loop

Formally, we use the notation `L`$_1$`(N)` to denote a loop $\mathsf{L}_1$ that has no nested loops within it, and its loop counter, say $\ell_1$, increases from 0 to an upper bound that is given by an expression in $N$. Similarly, we use `L`$_1$`(N, L`$_2$`(N))` to denote a loop $\mathsf{L}_1$ that has another loop $\mathsf{L}_2$ nested within it. The loop counter $\ell_1$ of $\mathsf{L}_1$ increases from 0 to an upper bound expression in $N$, while the loop counter $\ell_2$ of $\mathsf{L}_2$ increases from 0 to an upper bound expression in $\ell_1$ and $N$. Using this notation, `L`$_1$`(N, L`$_2$`(N, L`$_3$`(N)))` represents three nested loops, and so on. Notice that the upper bound expression for a nested loop can depend not only on $N$ but also on the loop counters of other loops nesting it. For notational clarity, we also use `LPeel(L`$_i$`, a, b)` to denote the peel of loop $\mathsf{L}_i$

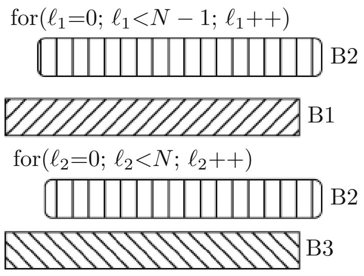

for($\ell_1$=0; $\ell_1$<$N-1$; $\ell_1$++)
B2
B1
for($\ell_2$=0; $\ell_2$<$N$; $\ell_2$++)
B2
B3

**Fig. 4.** Peel of the nested loop

consisting of all iterations of $\mathsf{L}_i$ where the value of $\ell_i$ ranges from `a` to `b-1`, both inclusive. Note that if `b-a` is a constant, this corresponds to the concatenation of `(b-a)` peels of $\mathsf{L}_i$.

We will now try to see how we can implement the transformation from the first column to the second column of Fig. 2 for a nested loop $\mathsf{L}_1$(N, $\mathsf{L}_2$(N)). The first step is to truncate all loops to use $N-1$ instead of $N$ in the upper bound expressions. Using the notation introduced above, this gives the loop $\mathsf{L}_1$(N-1, $\mathsf{L}_2$(N-1)). Note that all uses of $N$ other than in loop upper bound expressions stay unchanged as we go from $\mathsf{L}_1$(N, $\mathsf{L}_2$(N)) to $\mathsf{L}_1$(N-1, $\mathsf{L}_2$(N-1)). We now ask: *Which are the loop iterations of* $\mathsf{L}_1$(N, $\mathsf{L}_2$(N)) *that have been missed (or skipped) in going to* $\mathsf{L}_1$(N-1, $\mathsf{L}_2$(N-1))*?* Let the upper bound expression of $\mathsf{L}_1$ in $\mathsf{L}_1$(N, $\mathsf{L}_2$(N)) be $U_{\mathsf{L}_1}(N)$, and that of $\mathsf{L}_2$ be $U_{\mathsf{L}_2}(\ell_1, N)$. It is not hard to see that in every iteration $\ell_1$ of $\mathsf{L}_1$, where $0 \leq \ell_1 < U_{\mathsf{L}_1}(N-1)$, the iterations corresponding to $\ell_2 \in \{U_{\mathsf{L}_2}(\ell_1, N-1), \ldots, U_{\mathsf{L}_2}(\ell_1, N) - 1\}$ have been missed. In addition, all iterations of $\mathsf{L}_1$ corresponding to $\ell_1 \in \{U_{\mathsf{L}_1}(N-1), \ldots, U_{\mathsf{L}_1}(N) - 1\}$ have also been missed. This implies that the "peel" of $\mathsf{L}_1$(N, $\mathsf{L}_2$(N)) must include all the above missed iterations. This peel therefore is the program fragment shown in Fig. 5.

```
for(ℓ1=0; ℓ1<U_L1(N-1); ℓ1++)
  LPeel(L2, U_L2(ℓ1,N-1), U_L2(ℓ1,N))
LPeel(L1, U_L1(N-1), U_L1(N))
```

**Fig. 5.** Peel of $\mathsf{L}_1$(N, $\mathsf{L}_2$(N))

Notice that if $U_{\mathsf{L}_2}$ ($\ell_1$ ,N) - $U_{\mathsf{L}_2}$ ($\ell_1$ ,N-1) is a constant (as is the case if $U_{\mathsf{L}_2}$($\ell_1$,N) is any linear function of $\ell_1$ and $N$), then the peel does not have any loop with nesting depth 2. Hence, the maximum nesting depth of loops in the peel is strictly less than that in $\mathsf{L}_1$(N, $\mathsf{L}_2$(N)), yielding a peel that is "simpler" than the original program. This argument can be easily generalized to loops with arbitrarily large nesting depths. The peel of $\mathsf{L}_1$(N, $\mathsf{L}_2$(N, $\mathsf{L}_3$(N))) is as shown in Fig. 6.

```
for(ℓ1=0; ℓ1<U_L1(N-1); ℓ1++) {
  for(ℓ2=0; ℓ2<U_L2(ℓ1,N-1); ℓ2++)
    LPeel(L3, U_L3(ℓ1,ℓ2,N-1), U_L3(ℓ1,ℓ2,N))
  LPeel(L2, U_L2(ℓ1,N-1), U_L2(ℓ1,N))
}
LPeel(L1, U_L1(N-1), U_L1(N))
```

**Fig. 6.** Peel of $\mathsf{L}_1$(N, $\mathsf{L}_2$(N, $\mathsf{L}_3$(N)))

As an illustrative example, let us consider the program in Fig. 7(a), and suppose we wish to compute the peel of this program containing nested loops. In this case, the upper bounds of the loops are $U_{\mathsf{L}_1}(N) = U_{\mathsf{L}_2}(N) = N$. The peel is shown in Fig. 7(b) and consists of two sequentially composed non-nested loops. The first loop takes into account the missed iterations of the inner loop (a single iteration in this example) that are executed in $\mathsf{P}_N$ but are missed in $\mathsf{Q}_{N-1}$. The

```
for(i=0; i<N; i++)        for(i=0; i<N-1; i++)
  for(j=0; j<N; j++)        A[i][N-1] = N;
    A[i][j] = N;            for(j=0; j<N; j++)
                              A[N-1][j] = N;
          (a)                         (b)
```

**Fig. 7.** (a) Nested Loop & (b) Peel

**Algorithm 1.** GenQandPeel($\mathsf{P}_N$: program)

```
1: Let sequentially composed loops in $\mathsf{P}_N$ be in the order $\mathsf{L}_1$, $\mathsf{L}_2$, ..., $\mathsf{L}_m$;
2: for each loop $\mathsf{L}_i \in$ TopLevelLoops($\mathsf{P}_N$) do
3:     $\langle \mathsf{Q}_{\mathsf{L}_i}, \mathsf{R}_{\mathsf{L}_i} \rangle \leftarrow$ GenQandPeelForLoop($\mathsf{L}_i$);
4: while $\exists v.use(v) \in \mathsf{Q}_{\mathsf{L}_i} \wedge def(v) \in \mathsf{R}_{\mathsf{L}_j}$, for some $1 \le j < i \le N$ do    ▷ $v$ is var/array element
5:     Substitute rhs expression for $v$ from $\mathsf{R}_{\mathsf{L}_j}$ in $\mathsf{Q}_{\mathsf{L}_i}$;    ▷ If $\mathsf{R}_{\mathsf{L}_j}$ is a loop, abort
6: $\mathsf{Q}_{N-1} \leftarrow \mathsf{Q}_{\mathsf{L}_1}; \mathsf{Q}_{\mathsf{L}_2}; \ldots; \mathsf{Q}_{\mathsf{L}_m}$;
7: $\mathsf{peel}(\mathsf{P}_N) \leftarrow \mathsf{R}_{\mathsf{L}_1}; \mathsf{R}_{\mathsf{L}_2}; \ldots; \mathsf{R}_{\mathsf{L}_m}$;
8: return $\langle \mathsf{Q}_{N-1}, \mathsf{peel}(\mathsf{P}_N) \rangle$;

9: procedure GenQandPeelForLoop(L: loop)
10:     Let $U_\mathsf{L}(N)$ be the UB expression of loop L;
11:     $\mathsf{Q}_\mathsf{L} \leftarrow$ L with $N-1$ substituted for $N$ in all UB expressions (including for nested loops);
12:     if L has subloops then
13:         $t \leftarrow$ nesting depth of inner-most nested loop in L;
14:         $\mathsf{R}_{t+1} \leftarrow$ empty program with no statements;
15:         for $k = t; k \ge 2; k$-- do
16:             for each subloop $SL_j$ in $\mathsf{L}_i$ at nesting depth $k$ do    ▷ Ordered $SL_1$, $SL_2$, ..., $SL_j$
17:                 $\mathsf{R}_{SL_j} \leftarrow$ LPeel($SL_j$, $U_{SL_j}(\ell_1, \ldots, \ell_{k-1}, N-1)$, $U_{SL_j}(\ell_1, \ldots, \ell_{k-1}, N)$);
18:             $\mathsf{R}_k \leftarrow$ for (i=0; i<$U_{\mathsf{L}_{k-1}}(N-1)$; i++) { $\mathsf{R}_{k+1}$;$\mathsf{R}_{SL_1}$;$\mathsf{R}_{SL_2}$;...;$\mathsf{R}_{SL_j}$ };
19:         $\mathsf{R}_\mathsf{L} \leftarrow \mathsf{R}_2$;LPeel(L, $U_\mathsf{L}(N-1), U_\mathsf{L}(N)$);
20:     else
21:         $\mathsf{R}_\mathsf{L} \leftarrow$ LPeel(L, $U_\mathsf{L}(N-1), U_\mathsf{L}(N)$);
22:     return $\langle \mathsf{Q}_\mathsf{L}, \mathsf{R}_\mathsf{L} \rangle$;
```

second loop takes into account the missed iterations of the outer loop in $\mathsf{Q}_{N-1}$ compared to $\mathsf{P}_N$.

Generalizing the above intuition, Algorithm 1 presents function GenQandPeel for computing $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ for a given $\mathsf{P}_N$ that has sequentially composed loops with potentially nested loops. Due to the grammar of our programs, our loops are well nested. The method works by traversing over the structure of loops in the program. In this algorithm $\mathsf{Q}_{\mathsf{L}_i}$ and $\mathsf{R}_{\mathsf{L}_i}$ represent the counterparts of $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ for loop $\mathsf{L}_i$. We create the program $\mathsf{Q}_{N-1}$ by peeling each loop in the program and then propagating these peels across subsequent loops. We identify the missed iterations of each loop in the program $\mathsf{P}_N$ from the upper bound expression UB. Recall that the upper bound of each loop $\mathsf{L}_k$ at nesting depth $k$, denoted by $U_{\mathsf{L}_k}$ is in terms of the loop counters $\ell$ of outer loops and the program parameter $N$. We need to peel $U_{\mathsf{L}_k}(\ell_1, \ell_2, \ldots, \ell_{k-1}, N) - U_{\mathsf{L}_k}(\ell_1, \ell_2, \ldots, \ell_{k-1}, N-1)$ number of iterations from each loop, where $\ell_1 \le \ell_2 \le \ldots \le \ell_{k-1}$ are counters of the outer nesting loops. As discussed above, whenever this difference is a constant value, we are guaranteed that the loop nesting depth reduces by one. It may so happen that there are multiple sequentially composed loops $SL_j$ at nesting depth $k$ and not just a single loop $\mathsf{L}_k$. At line 2, we iterate over top level loops and call function GenQandPeelForLoop($\mathsf{L}_i$) for each sequentially composed loop $\mathsf{L}_i$ in $\mathsf{P}_N$. At line 11 we construct $\mathsf{Q}_\mathsf{L}$ for loop L. If the loop L has no nested loops, then the peel is the last iterations computed using the upper bound in line 21 For nested loops, the loop at line 15 builds the peel for all loops inside L following the above intuition. The peels of all sub-loops are collected and inserted in the peel of L at line 19. Since all the peeled iterations are moved after $Q_\mathsf{L}$ of each loop, we need to repair expressions appearing in $Q_{\mathsf{L}}$. The repairs are applied by the loop at line 4. In the repair step, we identify the right hand side expressions for all the variables and array elements assigned in the peeled iterations. Subsequently, the uses of the variables and arrays in $\mathsf{Q}_{\mathsf{L}_i}$ that are assigned in $\mathsf{R}_{\mathsf{L}_j}$ are replaced with the assigned expressions whenever $j < i$. If $\mathsf{R}_{\mathsf{L}_j}$ is a loop, this step is more involved and hence currently not considered. Finally at line 8, the peels and $Q$s of all top level loops are stitched and returned.

Note that lines 4 and 5 of Algorithm 1 implement the substitution represented by the arrow in the second column of Fig. 2. This is necessary in order to move the peel of a loop to the end of the program. If either of the loops $\mathsf{L}_i$ or $\mathsf{L}_j$ use array elements as index to other arrays then it can be difficult to identify what expression to use in $\mathsf{Q}_{\mathsf{L}_i}$ for the substitution. However, such scenarios are observed less often, and hence, they hardly impact the effectiveness of the technique on programs seen in practice. The peel $\mathsf{R}_{\mathsf{L}_j}$, from which the expression to be substituted in $\mathsf{Q}_{\mathsf{L}_i}$ has to be taken, itself may have a loop. In such cases, it can be significantly more challenging to identify what expression to use in $\mathsf{Q}_{\mathsf{L}_i}$. We use several optimizations to transform the peeled loop before trying to identify such an expression. If the modified values in the peel can be summarized as closed form expressions, then we can replace the loop in the peel with its summary. For example, consider the peeled loop, `for (` $\ell_1$ `=0;` $\ell_1 <$ `N;` $\ell_1$ `++) {` `S = S + 1; }`. This loop is summarized as `S = S + N;` before it can be moved across subsequent code. If the variables modified in the peel of a nested loop are not used later, then the peel can be trivially moved. In many cases, the loop in the peel can also be substituted with its conservative over-approximation. We have implemented some of these optimizations in our tool and are able to verify several benchmarks with sequentially composed nested loops. It may not always be possible to move the peel of a nested loop across subsequent loops but we have observed that these optimizations suffice for many programs seen in practice.

**Theorem 1.** *Let* $\mathsf{Q}_{N-1}$ *and* $\mathsf{peel}(\mathsf{P}_N)$ *be generated by application of function* GENQANDPEEL *from Algorithm 1 on program* $\mathsf{P}_N$*. Then* $\mathsf{P}_N$ *is semantically equivalent to* $\mathsf{Q}_{N-1}; \mathsf{peel}(\mathsf{P}_N)$*.*

**Lemma 1.** *Suppose the following conditions hold;*

- *Program* $\mathsf{P}_N$ *satisfies our syntactic restrictions (see Sect. 3).*
- *The upper bound expressions of all loops are linear expressions in* $N$ *and in the loop counters of outer nesting loops.*

*Then, the max nesting depth of loops in* $\mathsf{peel}(\mathsf{P}_N)$ *is strictly less than that in* $\mathsf{P}_N$*.*

*Proof.* Let $U_{\mathsf{L}_k}(\ell_1, \ldots, \ell_{k-1}, N)$ be the upper bound expression of a loop $\mathsf{L}_k$ at nesting depth $k$. Suppose $U_{\mathsf{L}_k} = c_1.\ell_1 + \cdots c_{k-1}.\ell_{k-1} + C.N + D$, where $c_1, \ldots c_{k-1}, C$ and $D$ are constants. Then $U_{\mathsf{L}_k}(\ell_1, \ldots, \ell_{k-1}, N) - U_{\mathsf{L}_k}(\ell_1, \ldots \ell_{k-1}, N-1) = C$, i.e. a constant. Now, recalling the discussion in Sect. 4.1, we see that `LPeel(`$\mathsf{L}_k$`,` $U_k$`(`$\ell_1,\ldots,\ell_{k-1}$`,N` $-$ `1),` $U_k$`(`$\ell_1,\ldots,\ell_{k-1}$`,N))` simply results in concatenating a constant number of peels of the loop $\mathsf{L}_k$. Hence, the maximum nesting depth of loops in $\texttt{LPeel(}\ \mathsf{L}_k,\ U_k\texttt{(}\ell_1,\ldots,\ell_{k-1}\texttt{,N}-\texttt{1),}$ $U_k\texttt{(}\ell_1,\ldots,\ell_{k-1}\texttt{,N))}$ is strictly less than the maximum nesting depth of loops in $\mathsf{L}_k$.

Suppose loop $\mathsf{L}$ with nested loops (having maximum nesting depth $t$) is passed as the argument of function GenQandPeelForLoop (see Algorithm 1). In line 15 of function GenQandPeelForLoop, we iterate over all loops at nesting depth 2 and above within $\mathsf{L}$. Let $\mathsf{L}_k$ be a loop at nesting depth $k$, where $2 \leq k \leq t$. Clearly, $\mathsf{L}_k$ can have at most $t-k$ nested levels of loops within it. Therefore, when `LPeel` is invoked on such a loop, the maximum nesting depth of loops in the peel generated for $\mathsf{L}_k$ can be at most $t-k-1$. From lines 18 and 19 of function GenQandPeelForLoop, we also know that this `LPeel` can itself appear at nesting depth $k$ of the overall peel $\mathsf{R}_\mathsf{L}$. Hence, the maximum nesting depth of loops in $\mathsf{R}_\mathsf{L}$ can be $t-k-1+k$, i.e. $t-1$. This is strictly less than the maximum nesting depth of loops in $\mathsf{L}$. $\square$

**Corollary 1.** *If* $\mathsf{P}_N$ *has no nested loops, then* $\mathsf{peel}(\mathsf{P}_N)$ *is loop-free.*

## 4.2 Generating $\varphi'(N-1)$ and $\Delta\varphi'(N)$

Given $\varphi(N)$, we check if it is of the form $\bigwedge_{i=0}^{N-1} \rho_i$, where $\rho_i$ is a formula on the $i^{th}$ elements of one or more arrays, and scalars used in $\mathsf{P}_N$. If so, we infer $\varphi'(N-1)$ to be $\bigwedge_{i=0}^{N-2} \rho_i$ and $\Delta\varphi'(N)$ to be $\rho_{N-1}$ (assuming variables/array elements in $\rho_{N-1}$ are not modified by $\mathsf{Q}_{N-1}$). Note that all uses of $N$ in $\rho_i$ are retained as is (i.e. not changed to $N-1$) in $\varphi'(N-1)$. In general, when deriving $\varphi'(N-1)$, we do not replace any use of $N$ in $\varphi(N)$ by $N-1$ unless it is the limit of an iterated conjunct as discussed above. Specifically, if $\varphi(N)$ doesn't contain an iterated conjunct as above, then we consider $\varphi'(N-1)$ to be the same as $\varphi(N)$ and $\Delta\varphi'(N)$ to be True. Thus, our generation of $\varphi'(N-1)$ and $\Delta\varphi'(N)$ differs from that of [12]. As discussed earlier, this makes it possible to reason about a much larger class of pre-conditions than that admissible by the technique of [12].

## 4.3 Inferring Inductive Difference Invariants

Once we have $\mathsf{P}_{N-1}$, $\mathsf{Q}_{N-1}$, $\varphi(N-1)$ and $\varphi'(N-1)$, we infer *difference invariants*. We construct the standard cross-product of programs $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$, denoted as $\mathsf{Q}_{N-1} \times \mathsf{P}_{N-1}$, and infer difference invariants at key control points. Note that $\mathsf{P}_{N-1}$ and $\mathsf{Q}_{N-1}$ are guaranteed to have synchronized iterations of corresponding loops (both are obtained by restricting the upper bounds of all loops to use $N-1$ instead of $N$). However, the conditional statements within the loop body may not be synchronized. Thus, whenever we can infer that the corresponding conditions are equivalent, we synchronize the branches of the conditional statement. Otherwise, we consider all four possibilities of the branch conditions. It can be seen that the net effect of the cross-product is executing the programs $\mathsf{P}_{N-1}$ and $\mathsf{Q}_{N-1}$ one after the other.

We run a dataflow analysis pass over the constructed product graph to infer difference invariants at loop head, loop exit and at each branch condition. The only dataflow values of interest are differences between corresponding variables in $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$. Indeed, since structure and variables of $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$ are similar, we can create the correspondence map between the variables. We start the difference invariant generation by considering relations between corresponding variables/array elements appearing in pre-conditions of the two programs. We apply static analysis that can track equality expressions (including disjunctions over equality expressions) over variables as we traverse the program. These equality expressions are our difference invariants.

We observed in our experiments the most of the inferred equality expressions are simple expressions of $N$ (atmost quadratic in $N$). This not totally surprising and similar observations have also been independently made in [4,15,24]. Note that the difference invariants may not always be equalities. We can easily extend our analysis to learn inequalities using interval domains in static analysis. We can also use a library of expressions to infer difference invariants using a guess-and-check framework. Moreover, guessing difference invariants can be easy as in many cases the difference expressions may be independent of the program constructs, for example, the equality expression $\mathtt{v} = \mathtt{v}'$ where $\mathtt{v} \in \mathsf{P}_{N-1}$ and $\mathtt{v}' \in \mathsf{Q}_{N-1}$ does not depend on any other variable from the two programs.

For the example in Fig. 2, the difference invariant at the head of the first loop of $Q_{N-1} \times P_{N-1}$ is $D(V_\mathsf{Q}, V_\mathsf{P}, N-1) \equiv (\mathtt{x}' - \mathtt{x} = \mathtt{i} \times (2 \times \mathtt{N} - 1) \wedge \forall \mathtt{i} \in [0, \mathtt{N}-1),\ \mathtt{a}'[\mathtt{i}] - \mathtt{a}[\mathtt{i}] = 1)$, where $\mathtt{x}, \mathtt{a} \in V_\mathsf{P}$ and $\mathtt{x}', \mathtt{a}' \in V_\mathsf{Q}$. Given this, we easily get $\mathtt{x}' - \mathtt{x} = (\mathtt{N}-1) \times (2 \times \mathtt{N} - 1)$ when the first loop terminates. For the second loop, $D(V_\mathsf{Q}, V_\mathsf{P}, N-1) \equiv (\forall \mathtt{j} \in [0, \mathtt{N}-1),\ \mathtt{b}'[\mathtt{j}] - \mathtt{b}[\mathtt{j}] = (\mathtt{x}' - \mathtt{x}) + \mathtt{N}^2 = (\mathtt{N}-1) \times\ (2 \times \mathtt{N} - 1) + \mathtt{N}^2)$.

Note that the difference invariants and its computation are agnostic of the given post-condition. Hence, our technique does not need to re-run this analysis for proving a different post-condition for the same program.

## 4.4 Verification Using Inductive Difference Invariants

We present our method DIFFY for verification of programs using inductive difference invariants in Algorithm 2. It takes a Hoare triple $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ as input, where $\varphi(N)$ and $\psi(N)$ are pre- and post-condition formulas. We check the base in line 1 to verify the Hoare triple for $N = 1$. If this check fails, we report a counterexample. Subsequently, we compute $\mathsf{Q}_{N-1}$ and $\mathsf{peel}(\mathsf{P}_N)$ as described in Sect. 4.1 using the function GENQANDPEEL from Algorithm 1. At line 4, we compute the formulas $\varphi'(N-1)$ and $\Delta\varphi'(N)$ as described in Sect. 4.2. For automation, we analyze the quantifiers appearing in $\varphi(N)$ and modify the quantifier ranges such that the conditions in Sect. 4.2 hold. We infer difference invariants $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ on line 5 using the method described in Sect. 4.3, wherein $V_\mathsf{Q}$ and $V_\mathsf{P}$ are sets of variables from $\mathsf{Q}_{N-1}$ and $\mathsf{P}_{N-1}$ respectively. At line 6, we compute $\psi'(N-1)$ by eliminating variables $V_\mathsf{P}$ from $\mathsf{P}_{N-1}$ from $\psi(N-1) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$. At line 7, we check the inductive step of our analysis. If the inductive step succeeds, then we conclude that the assertion holds.

**Algorithm 2.** Diffy( $\{\varphi(N)\}$ $\mathsf{P}_N$ $\{\psi(N)\}$ )

```
1: if $\{\varphi(1)\}$ $\mathsf{P}_1$ $\{\psi(1)\}$ fails then                              ▷ Base case for N=1
2:     return "Counterexample found!";

3: $\langle \mathsf{Q}_{N-1}, \mathsf{peel}(\mathsf{P}_N)\rangle \leftarrow$ GenQandPeel($\mathsf{P}_N$);
4: $\langle \varphi'(N-1), \Delta\varphi'(N)\rangle \leftarrow$ FormulaDiff($\varphi(N)$);      ▷ $\varphi(N) \Rightarrow \varphi'(N-1) \wedge \Delta\varphi'(N)$
5: $D(V_\mathsf{Q}, V_\mathsf{P}, N-1) \leftarrow$ InferDiffInvs($\mathsf{Q}_{N-1}, \mathsf{P}_{N-1}, \varphi'(N-1), \varphi(N-1)$);
6: $\psi'(N-1) \leftarrow$ QE($V_\mathsf{P}, \psi(N-1) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$);
7: if $\{\psi'(N-1) \wedge \Delta\varphi'(N)\}$ $\mathsf{peel}(\mathsf{P}_N)$ $\{\psi(N)\}$ then
8:     return True;                                                            ▷ Verification Successful
9: else
10:     return Strengthen($\mathsf{P}_N, \mathsf{peel}(\mathsf{P}_N), \varphi(N), \psi(N), \psi'(N-1), \Delta\varphi'(N), D(V_\mathsf{Q}, V_\mathsf{P}, N)$);

11: procedure Strengthen($\mathsf{P}_N$, $\mathsf{peel}(\mathsf{P}_N)$, $\varphi(N)$, $\psi(N)$, $\psi'(N-1)$, $\Delta\varphi'(N)$, $D(V_\mathsf{Q}, V_\mathsf{P}, N)$)
12:     $\chi(N) \leftarrow \psi(N)$;
13:     $\xi(N) \leftarrow$ True;
14:     $\xi'(N-1) \leftarrow$ True;
15:     repeat
16:         $\chi'(N-1) \leftarrow$ WP($\chi(N), \mathsf{peel}(\mathsf{P}_N)$);          ▷ Dijkstra's WP for loop free code
17:         if $\chi'(N-1) = \emptyset$ then
18:             if $\mathsf{peel}(\mathsf{P}_N)$ has a loop then
19:                 return Diffy($\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{peel}(\mathsf{P}_N)$ $\{\xi(N) \wedge \psi(N)\}$);
20:             else
21:                 return False;                                              ▷ Unable to prove
22:         $\chi(N) \leftarrow$ QE($V_\mathsf{Q}, \chi'(N) \wedge D(V_\mathsf{Q}, V_\mathsf{P}, N)$);
23:         $\xi(N) \leftarrow \xi(N) \wedge \chi(N)$;
24:         $\xi'(N-1) \leftarrow \xi'(N-1) \wedge \chi'(N-1)$;
25:         if $\{\varphi(1)\}$ $\mathsf{P}_1$ $\{\xi(1)\}$ fails then
26:             return False;                                                  ▷ Unable to prove
27:         if $\{\xi'(N-1) \wedge \Delta\varphi'(N) \wedge \psi'(N-1)\}$ $\mathsf{peel}(\mathsf{P}_N)$ $\{\xi(N) \wedge \psi(N)\}$ holds then
28:             return True;                                                   ▷ Verification Successful
29:     until timeout;
30:     return False;
```

If that is not the case then, we try to iteratively strengthen both the pre- and post-condition of $\mathsf{peel}(\mathsf{P}_N)$ simultaneously by invoking Strengthen.

The function Strengthen first initializes the formula $\chi(N)$ with $\psi(N)$ and the formulas $\xi(N)$ and $\xi'(N-1)$ to True. To strengthen the pre-condition of $\mathsf{peel}(\mathsf{P}_N)$, we infer a formula $\chi'(N-1)$ using Dijkstra's weakest pre-condition computation of $\chi(N)$ over the $\mathsf{peel}(\mathsf{P}_N)$ in line 16. It may happen that we are unable to infer such a formula. In such a case, if the program $\mathsf{peel}(\mathsf{P}_N)$ has loops then we recursively invoke Diffy at line 19 to further simplify the program. Otherwise, we abandon the verification effort (line 21). We use quantifier elimination to infer $\chi(N-1)$ from $\chi'(N-1)$ and $D(V_\mathsf{Q}, V_\mathsf{P}, N-1))$ at line 6.

The inferred pre-conditions $\chi(N)$ and $\chi'(N-1)$ are accumulated in $\xi(N)$ and $\xi'(N-1)$, which strengthen the post-conditions of $\mathsf{P}_N$ and $\mathsf{Q}_{N-1}$ respectively in lines 23–24. We again check the base case for the inferred formulas in $\xi(N)$ at line 25. If the check fails we abandon the verification attempt at line 26. If the base case succeeds, we then proceed to the inductive step. When the inductive step succeeds, we conclude that the assertion is verified. Otherwise, we continue in the loop and try to infer more pre-conditions untill we run out of time.

The pre-condition in Fig. 2 is $\phi(N) \equiv$ True and the post-condition is $\psi(N) \equiv \forall \mathtt{j} \in [\mathtt{0}, \mathtt{N}),\ \mathtt{b}[\mathtt{j}] = \mathtt{j} + \mathtt{N}^3)$. At line 4, $\phi'(N-1)$ and $\Delta\phi'(N-1)$ are computed to be True. $D(V_\mathsf{Q}, V_\mathsf{P}, N-1)$ is the formula computed in Sect. 4.3. At line 6,

**Table 1.** Summary of the experimental results. S is successful result. U is inconclusive result. TO is timeout.

| PROGRAM CATEGORY | | DIFFY | | | VAJRA | | VERIABS | | VIAP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | S | U | TO | S | U | S | TO | S | U | TO |
| Safe C1 | 110 | 110 | 0 | 0 | 110 | 0 | 96 | 14 | 16 | 1 | 93 |
| Safe C2 | 24 | 21 | 0 | 3 | 0 | 24 | 5 | 19 | 4 | 0 | 20 |
| Safe C3 | 23 | 20 | 3 | 0 | 0 | 23 | 9 | 14 | 0 | 23 | 0 |
| Total | 157 | 151 | 3 | 3 | 110 | 47 | 110 | 47 | 20 | 24 | 113 |
| Unsafe C1 | 99 | 98 | 1 | 0 | 98 | 1 | 84 | 15 | 98 | 0 | 1 |
| Unsafe C2 | 24 | 24 | 0 | 0 | 17 | 7 | 19 | 5 | 22 | 0 | 2 |
| Unsafe C3 | 23 | 20 | 3 | 0 | 0 | 23 | 22 | 1 | 0 | 23 | 0 |
| Total | 146 | 142 | 4 | 0 | 115 | 31 | 125 | 21 | 120 | 23 | 3 |

$\psi'(N-1) \equiv (\forall \mathtt{j} \in [\mathtt{0}, \mathtt{N}-\mathtt{1}),\ \mathtt{b}'[\mathtt{j}] = \mathtt{j} + (\mathtt{N}-\mathtt{1})^3 + (\mathtt{N}-\mathtt{1}) \times (\mathtt{2} \times \mathtt{N} - \mathtt{1}) + \mathtt{N}^2 = \mathtt{j} + \mathtt{N}^3)$. The algortihm then invokes STRENGTHEN at line 10 which infers the formulas $\chi'(N-1) \equiv (\mathtt{x}' = (\mathtt{N}-\mathtt{1})^3)$ at line 16 and $\chi(N) \equiv (\mathtt{x} = \mathtt{N}^3)$ at line 22. These are accumulated in $\xi'(N-1)$ and $\xi(N)$, simultaneosuly strengthening the pre- and post-condition. Verification succeeds after this strengthening iteration.

The following theorem guarantees the soundness of our technique.

**Theorem 2.** *Suppose there exist formulas $\xi'(N)$ and $\xi(N)$ and an integer $M > 0$ such that the following hold*

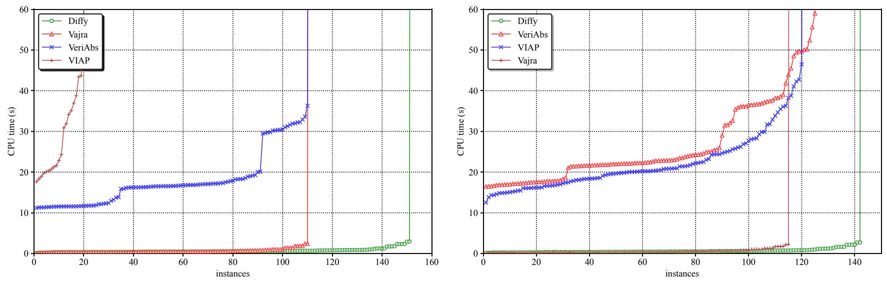
– $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\ \wedge\ \xi(N)\}$ *holds for* $1 \leq N \leq M$*, for some* $M > 0$*.*
– $\xi(N) \wedge D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N) \Rightarrow \xi'(N)$ *for all* $N > 0$*.*
– $\{\xi'(N-1)\ \wedge\ \Delta\varphi'(N)\ \ \wedge\ \ \psi'(N-1)\}\ \mathsf{peel}(\mathsf{P}_N)\ \{\xi(N)\ \wedge\ \psi(N)\}$ *holds for all* $N \geq M$*, where* $\psi'(N-1) \equiv \exists V_{\mathsf{P}}\big(\psi(N-1)\ \wedge\ D(V_{\mathsf{Q}}, V_{\mathsf{P}}, N-1)\big)$*.*

*Then* $\{\varphi(N)\}\ \mathsf{P}_N\ \{\psi(N)\}$ *holds for all* $N > 0$*.*

## 5 Experimental Evaluation

We have instantiated our technique in a prototype tool called DIFFY. It is written in `C++` and is built using the LLVM(v6.0.0) [31] compiler. We use the SMT solver Z3(v4.8.7) [39] for proving Hoare triples of loop-free programs. DIFFY and the supporting data to replicate the experiments are openly available at [14].

**Setup.** All experiments were performed on a machine with Intel i7-6500U CPU, 16 GB RAM, running at 2.5 GHz, and Ubuntu 18.04.5 LTS operating system. We have compared the results obtained from DIFFY with VAJRA(v1.0) [12], VIAP(v1.1) [42] and VERIABS(v1.4.1-12) [1]. We choose VAJRA which also employs inductive reasoning for proving array programs and verify the benchmarks in its test-suite. We compared with VERIABS as it is the winner of the arrays sub-category in SV-COMP 2020 [6] and 2021 [7]. VERIABS applies a



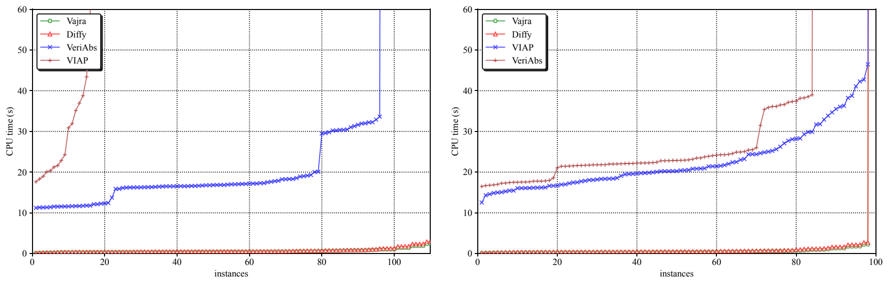
(a) (b)

**Fig. 8.** Cactus Plots (a) All Safe Benchmarks (b) All Unsafe Benchmarks

sequence of techniques from its portfolio to verify array programs. We compared with VIAP which was the winner in arrays sub-category in SV-COMP 2019 [5]. VIAP also employs a sequence of tactics, implemented for proving a variety of array programs. Diffy does not use multiple techniques, however we choose to compare it with these portfolio verifiers to show that it performs well on a class of programs and can be a part of their portfolio. All tools take C programs in the SV-COMP format as input. Timeout of 60 s was set for each tool. A summary of the results is presented in Table 1.

**Benchmarks.** We have evaluated Diffy on a set of 303 array benchmarks, comprising of the entire test-suite of [12], enhanced with challenging benchmarks to test the efficacy of our approach. These benchmarks take a symbolic parameter $N$ which specifies the size of each array. Assertions are (in-)equalities over array elements, scalars and (non-)linear polynomial terms over $N$. We have divided both the safe and unsafe benchmarks in three categories. Benchmarks in C1 category have standard array operations such as min, max, init, copy, compare as well as benchmarks that compute polynomials. In these benchmarks, branch conditions are not affected by the value of $N$, operations such as modulo and nested loops are not present. There are 110 safe and 99 unsafe programs in the C1 category in our test-suite. In C2 category, the branch conditions are affected by change in the program parameter $N$ and operations such as modulo are used in these benchmarks. These benchmarks do not have nested loops in them. There are 24 safe and unsafe benchmarks in the C2 category. Benchmarks in category C3 are programs with atleast one nested loop in them. There are 23 safe and unsafe programs in category C3 in our test-suite. The test-suite has a total of 157 safe and 146 unsafe programs.

**Analysis.** Diffy verified 151 safe benchmarks, compared to 110 verified by Vajra as well as VeriAbs and 20 verified by VIAP. Diffy was unable to verify 6 safe benchmarks. In 3 cases, the smt solver timed out while trying to prove the induction step since the formulated query had a modulus operation and in 3 cases it was unable to compute the predicates needed to prove the assertions. Vajra was unable to verify 47 programs from categories C2 and



(a) (b)

**Fig. 9.** Cactus plots (a) Safe C1 benchmarks (b) Unsafe C1 benchmarks

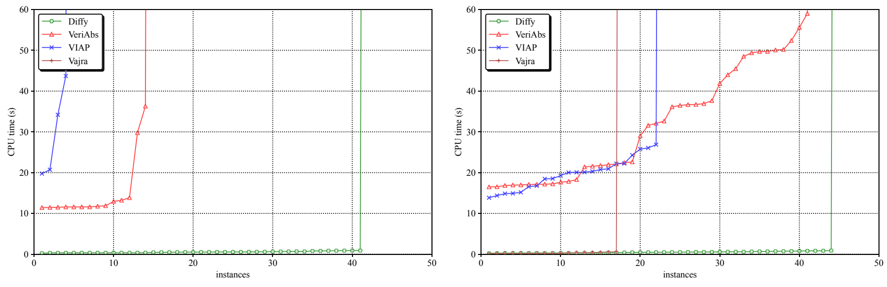
C3. These are programs with nested loops, branch conditions affected by $N$, and cases where it could not compute the difference program. The sequence of techniques employed by VERIABS, ran out of time on 47 programs while trying to prove the given assertion. VERIABS proved 2 benchmarks in category C2 and 3 benchmarks in category C3 where DIFFY was inconclusive or timed out. VERIABS spends considerable amount of time on different techniques in its portfolio before it resorts to VAJRA and hence it could not verify 14 programs that VAJRA was able to prove efficiently. VIAP was inconclusive on 24 programs which had nested loops or constructs that could not be handled by the tool. It ran out of time on 113 benchmarks as the initial tactics in its sequence took up the allotted time but could not verify the benchmarks. DIFFY was able to verify all programs that VIAP and VAJRA were able to verify within the specified time limit.

The cactus plot in Fig. 8(a) shows the performance of each tool on all safe benchmarks. DIFFY was able to prove most of the programs within three seconds. The cactus plot in Fig. 9(a) shows the performance of each tool on safe benchmarks in C1 category. VAJRA and DIFFY perform equally well in the C1 category. This is due to the fact that both tools perform efficient inductive reasoning. DIFFY outperforms VERIABS and VIAP in this category. The cactus plot in Fig. 10(a) shows the performance of each tool on safe benchmarks in the combined categories C2 and C3, that are difficult for VAJRA as most of these programs are not within its scope. DIFFY out performs all other tools in categories C2 and C3. VERIABS was an order of magnitude slower on programs it was able to verify, as compared to DIFFY. VERIABS spends significant amount of time in trying techniques from its portfolio, including VAJRA, before one of them succeeds in verifying the assertion or takes up the entire time allotted to it. VIAP took 70 seconds more on an average as compared to DIFFY to verify the given benchmark. VIAP also spends a large portion of time in trying different tactics implemented in the tool and solving the recurrence relations in programs.

Our technique reports property violations when the base case of the analysis fails for small fixed values of $N$. While the focus of our work is on proving assertions, we report results on unsafe versions of the safe benchmarks from our test-suite. Diffy was able to detect a property violation in 142 unsafe programs and was inconclusive on 4 benchmarks. Vajra detected violations in 115 programs and was inconclusive on 31 programs. VeriAbs reported 125 programs as unsafe and ran out of time on 21 programs. VIAP reported property violation in 120 programs, was inconclusive on 23 programs and timed out on 3 programs.

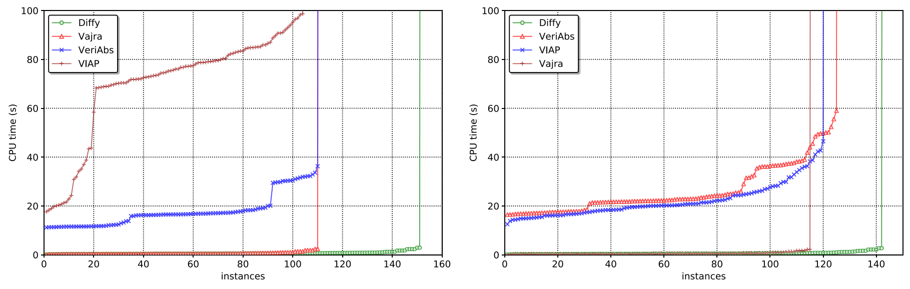
The cactus plot in Fig. 8(b) shows the performance of each tool on all unsafe benchmarks. Diffy was able to detect a violation faster than all other tools and on more benchmarks from the test-suite. Figure 9(b) and Fig. 10(b) give a finer glimpse of the performance of these tools on the categories that we have defined. In the C1 category, Diffy and Vajra have comparable performance and Diffy disproves the same number of benchmarks as Vajra and VIAP. In C2 and C3 categories, we are able to detect property violations in more benchmarks than other tools in less time.

To observe any changes in the performance of these, we also ran them with an increased time out of 100 seconds (Fig. 11). Performance remains unchanged for Diffy, Vajra and VeriAbs on both safe and unsafe benchmarks, and of VIAP on unsafe benchmarks. VIAP was able to additionally verify 89 safe programs in categories C1 and C2 with the increased time limit.



(a) (b)

**Fig. 10.** Cactus plots (a) Safe C2 & C3 benchmarks (b) Unsafe C2 & C3 benchmarks



(a) (b)

**Fig. 11.** Cactus plots. TO = 100 s. (a) Safe benchmarks (b) Unsafe benchmarks

## 6 Related Work

*Techniques Based on Induction.* Our work is related to several efforts that apply inductive reasoning to verify properties of array programs. Our work subsumes the full-program induction technique in [12] that works by inducting on the entire program via a program parameter $N$. We propose a principled method for computation and use of difference invariants, instead of computing difference programs which is more challenging. An approach to construct safety proofs by automatically synthesizing squeezing functions that shrink program traces is proposed in [27]. Such functions are not easy to synthesize, whereas difference invariants are relatively easy to infer. In [11], the post-condition is inductively established by identifying a tiling relation between the loop counter and array indices used in the program. Our technique can verify programs from [11], when supplied with the *tiling* relation. [44] identifies recurrent program fragments for induction using the loop counter. They require restrictive data dependencies, called *commutativity of statements*, to move peeled iterations across subsequent loops. Unfortunately, these restrictions are not satisfied by a large class of programs in practice, where our technique succeeds.

*Difference Computation.* Computing differences of program expressions has been studied for incremental computation of expensive expressions [35,41], optimizing programs with arrays [34], and checking data-structure invariants [45]. These differences are not always well suited for verifying properties, in contrast with the difference invariants which enable inductive reasoning in our case.

*Logic Based Reasoning.* In [21], trace logic that implicitly captures inductive loop invariants is described. They use theorem provers to introduce and prove lemmas at arbitrary control locations in the program. Unlike their technique, we focus primarily on universally quantified and quantifier-free properties, although a restricted class of existentially quantified properties can be handled by our technique (see [13] for more details). VIAP [42] translates the program to an quantified first-order logic formula using the scheme proposed in [32]. It uses a portfolio of tactics to simplify and prove the generated formulas. Dedicated solvers for recurrences are used whereas our technique adapts induction for handling recurrences.

*Invariant Generation.* Several techniques generate invariants for array programs. QUIC3 [25], FreqHorn [9,19] infer universally quantified invariants over arrays for Constrained Horn Clauses (CHCs). Template-based techniques [8,23,47] search for inductive quantified invariants by instantiating parameters of a fixed set of templates. We generate relational invariants, which are often easier to infer compared to inductive quantified invariants for each loop.

*Abstraction-Based Techniques.* Counterexample-guided abstraction refinement using prophecy variables for programs with arrays is proposed in [36]. VERIABS [1] uses a portfolio of techniques, specifically to identify loops that can be soundly abstracted by a bounded number of iterations. VAPHOR [38] transforms array programs to array-free Horn formulas to track bounded number of array cells. BOOSTER [3] combines lazy abstraction based interpolation [2] and acceleration [10,28] for array programs. Abstractions in [16,18,22,26,29,33,37] implicitly or explicitly partition the range array indices to infer and prove facts on array segments. In contrast, our method does not rely on abstractions.

## 7 Conclusion

We presented a novel verification technique that combines generation of difference invariants and inductive reasoning. Difference invariants relate corresponding variables and arrays from the two versions of a program and are often easy to infer and prove. We have instantiated these techniques in our prototype Diffy. Experiments shows that Diffy out-performs the tools that won the Arrays sub-category in SV-COMP 2019, 2020 and 2021. Although we have focused on universal and quantifier-free properties in this paper, the technique applies to some classes of existential properties as well. The interested reader is referred to [13] for more details. Investigations in using synthesis techniques for automatic generation of difference invariants to verify properties of array manipulating programs is a part of future work.

## References

1. Afzal, M., et al.: Veriabs: verification by abstraction and test generation (competition contribution). In: TACAS 2020. LNCS, vol. 12079, pp. 383–387. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_25
2. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: Bjørner, N., Voronkov, A. (eds.) LPAR 2012. LNCS, vol. 7180, pp. 46–61. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28717-6_7
3. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: an acceleration-based verification framework for array programs. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 18–23. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_2
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
5. Beyer, D.: Competition on software verification (SV-COMP) (2019). http://sv-comp.sosy-lab.org/2019/
6. Beyer, D.: Competition on software verification (SV-COMP) (2020). http://sv-comp.sosy-lab.org/2020/
7. Beyer, D.: Competition on software verification (SV-COMP) (2021). http://sv-comp.sosy-lab.org/2021/
8. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69738-1_27
9. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_8

10. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_23
11. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs by tiling. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 428–449. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_21
12. Chakraborty, S., Gupta, A., Unadkat, D.: Verifying array manipulating programs with full-program induction. In: TACAS 2020. LNCS, vol. 12078, pp. 22–39. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_2
13. Chakraborty, S., Gupta, A., Unadkat, D.: Diffy: inductive reasoning of array programs using difference invariants (2021). https://arxiv.org/abs/2105.14748
14. Chakraborty, S., Gupta, A., Unadkat, D.: Diffy: inductive reasoning of array programs using difference invariants, April 2021. https://doi.org/10.6084/m9.figshare.14509467
15. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proceedings of PLDI, pp. 1027–1040 (2019)
16. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of POPL, pp. 105–118 (2011)
17. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 127–147. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_7
18. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: beyond strong vs. weak updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_14
19. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 259–277. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_14
20. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proceedings of POPL, pp. 191–202 (2002)
21. Georgiou, P., Gleiss, B., Kovács, L.: Trace logic for inductive loop reasoning. In: Proceedings of FMCAD, pp. 255–263 (2020)
22. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: Proceedings of POPL, pp. 338–350 (2005)
23. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proceedings of POPL, pp. 235–246 (2008)
24. Gupta, S., Rose, A., Bansal, S.: Counterexample-guided correlation algorithm for translation validation. Proc. OOPSLA **4**, 1–29 (2020)
25. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 248–266. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_15
26. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proceedings of PLDI, pp. 339–348 (2008)
27. Ish-Shalom, O., Itzhaky, S., Rinetzky, N., Shoham, S.: Putting the squeeze on array programs: loop verification via inductive rank reduction. In: Proceedings of VMCAI, pp. 112–135 (2020)
28. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: Proceedings of POPL, pp. 529–540 (2014)

29. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_23
30. Knobe, K., Sarkar, V.: Array ssa form and its use in parallelization. In: Proceedings of POPL, pp. 107–120 (1998)
31. Lattner, C.: LLVM and clang: next generation compiler technology. In: The BSD Conference, pp. 1–2 (2008)
32. Lin, F.: A formalization of programs in first-order logic with a discrete linear order. Artif. Intell. **235**, 1–25 (2016)
33. Liu, J., Rival, X.: Abstraction of arrays based on non contiguous partitions. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) VMCAI 2015. LNCS, vol. 8931, pp. 282–299. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46081-8_16
34. Liu, Y.A., Stoller, S.D., Li, N., Rothamel, T.: Optimizing aggregate array computations in loops. TOPLAS **27**(1), 91–125 (2005)
35. Liu, Y.A., Stoller, S.D., Teitelbaum, T.: Static caching for incremental computation. TOPLAS **20**(3), 546–585 (1998)
36. Mann, M., Irfan, A., Griggio, A., Padon, O., Barrett, C.: Counterexample-guided prophecy for model checking modulo the theory of arrays. In: TACAS 2021. LNCS, vol. 12651, pp. 113–132. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_7
37. Monniaux, D., Alberti, F.: A simple abstraction of arrays and maps by program translation. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 217–234. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_13
38. Monniaux, D., Gonnord, L.: Cell morphing: from array programs to array-free horn clauses. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 361–382. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_18
39. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
40. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of PLDI, pp. 83–94 (2000)
41. Paige, R., Koenig, S.: Finite differencing of computable expressions. TOPLAS **4**(3), 402–454 (1982)
42. Rajkhowa, P., Lin, F.: Extending VIAP to handle array programs. In: Piskac, R., Rümmer, P. (eds.) VSTTE 2018. LNCS, vol. 11294, pp. 38–49. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03592-1_3
43. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of POPL, pp. 12–27 (1988)
44. Seghir, M.N., Brain, M.: Simplifying the verification of quantified array assertions via code transformation. In: Albert, E. (ed.) LOPSTR 2012. LNCS, vol. 7844, pp. 194–212. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38197-3_13
45. Shankar, A., Bodik, R.: Ditto: automatic incrementalization of data structure invariant checks (in Java). ACM SIGPLAN Not. **42**(6), 310–319 (2007)
46. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of OOPSLA, pp. 391–406 (2013)
47. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. ACM SIGPLAN Not. **44**(6), 223–234 (2009)

48. Zaks, A., Pnueli, A.: CoVaC: compiler validation by program analysis of the cross-product. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_5
49. Zuck, L., Pnueli, A., Fang, Y., Goldberg, B.: VOC: a translation validator for optimizing compilers. ENTCS **65**(2), 2–18 (2002)

# Author Index

Abate, Alessandro II-3
Agarwal, Pratyush I-341
Akshay, S. I-619
Albert, Elvira II-863
Alur, Rajeev I-249
Amram, Gal I-870
André, Étienne I-552
Andriushchenko, Roman I-856
Arcaini, Paolo I-595
Armstrong, Alasdair I-303
Arquint, Linard I-367
Ayoun, Sacha-Élie II-827

Backes, J. II-851
Bae, Kyungmin I-491
Baier, Christel I-894
Baier, Daniel II-195
Bak, Stanley I-263
Balunovic, Mislav I-225
Bansal, Suguman I-870
Bardin, Sébastien I-669
Barrett, Clark II-461
Batz, Kevin II-524
Baumeister, Jan I-694
Bayless, S. II-851
Bendík, Jaroslav II-313
Beneš, Nikola I-505
Berzish, Murphy II-289
Beyer, Dirk II-195
Biere, Armin II-363
Bodeveix, Jean-Paul II-337
Bonakdarpour, Borzoo I-694
Boston, Brett I-645
Bozzano, Marco II-209
Bragg, Nate F. F. I-808
Breese, Samuel I-645
Brim, Luboš I-505
Brown, Kristopher II-461
Brunel, Julien II-337

Campbell, Brian I-303
Carpenter, Taylor I-249
Cauli, Claudia I-767
Češka, Milan I-856
Chakraborty, Supratik II-911
Chalupa, Marek II-887
Chatterjee, Krishnendu I-341
Chemouil, David II-337
Chen, Guangke I-175
Chen, Jiayu I-225
Chen, Mingshuai I-443, II-524
Chen, Taolue I-175
Chen, Xiaohong II-477
Chiari, Michele II-387
Christakis, Maria I-201, II-777
Cimatti, Alessandro I-529, II-209
Clochard, Martin I-367
Coenen, Norine I-694, I-894
Cogumbreiro, Tiago I-403
Constantinides, George II-626
Cyphert, John I-46, I-783

D'Antoni, Loris I-84, I-783
DaCosta, D. II-851
Dahlqvist, Fredrik II-626
Dan, Andrei I-225
Day, Joel D. II-289
Dodds, Joey I-645
Dodds, Mike I-645
Dutertre, Bruno II-266
Dwyer, Matthew B. I-137

Eilers, Marco I-718
Eisenhut, Jan II-411
Elad, Neta I-317
Elbaum, Sebastian I-137
Eniser, Hasan Ferit I-201

Fang, Wang I-151
Farinier, Benjamin I-669
Farzan, Azadeh I-832
Ferlez, James I-287
Fernandes Pires, Anthony II-209
Finkbeiner, Bernd I-694, I-894
Foster, Jeffrey S. I-808
Fried, Dror I-870
Friedberger, Karlheinz II-195

Fu, Yu-Fu II-149
Funke, Florian I-894

Ganesh, Vijay II-289
Gardner, Philippa II-827
Gastin, Paul I-619
Genaim, Samir II-863
Giacobbe, Mirco II-3
Girol, Guillaume I-669
Gnad, Daniel II-411
Goel, Shilpi I-26
Gopinath, Divya I-3
Griggio, Alberto I-529, II-209
Guan, Ji I-151
Gupta, Aarti II-461
Gupta, Ashutosh II-911

Hahn, Ernst Moritz II-651
Hallé, Sylvain II-500
Hamilton, Nathaniel I-263
Hasuo, Ichiro I-595, II-75
Hauptman, Dustin I-566
Heljanko, Keijo II-363
Hermanns, Holger I-201
Hobor, Aquinas II-801
Hoffmann, Jörg I-201, II-411
Holtzen, Steven II-577
Hu, Qinheping I-84, I-783
Huffman, Brian I-645
Hur, Chung-Kil II-752

Immerman, Neil I-317
Irfan, Ahmed I-529, II-461
Itzhaky, Shachar I-110, II-125
Ivanov, Radoslav I-249

Jacobs, Bart II-27
Jacobs, Swen II-435
Jansen, Nils II-602
Jantsch, Simon I-894
Jewell, K. II-851
Johnson, Andrew I-380
Johnson, Taylor T. I-263
Jonáš, Martin II-209
Jones, B. F. II-851
Joshi, S. II-851
Jovanović, Dejan II-266
Junges, Sebastian I-856, II-553, II-577, II-602

Kaminski, Benjamin Lucien II-524
Katoen, Joost-Pieter I-443, I-856, II-524
Keshmiri, Shawn I-566
Khedr, Haitham I-287
Kim, Dongjoo II-752
Kim, Jinwoo I-84
Kim, Sharon I-491
Kimberly, Greg II-209
Kincaid, Zachary I-46, II-51
Klaška, David II-887
Kokologiannakis, Michalis I-427
Koskinen, Eric I-742
Kothari, Yugesh I-201
Kovács, Laura I-317
Kremer, Gereon II-231
Kulczynski, Mitja II-289
Kura, Satoshi II-75

Lal, Ratan I-566
Lange, Julien I-403
Launchbury, N. II-851
Lee, Insup I-249
Lee, Jaehun I-491
Lee, Juneyoung II-752
Lefaucheux, Engel II-172
Leow, Wei Xiang II-801
Leutgeb, Lorenz II-99
Li, Jianlin I-201
Li, Meng I-767
Li, Pengfei II-728
Li, Yangge I-580
Lin, Anthony W. II-243
Lin, Wang I-467
Lin, Zhengyao II-477
Liu, Jiaxiang II-149
Liu, Zhiming I-467
Lluch Lafuente, Alberto II-411
Lonsing, Florian II-461
Lopes, Nuno P. II-752
Lopez, Diego Manzanas I-263
Lyu, Deyun I-595

Ma, Lei I-595
Maksimović, Petar II-827
Mandrioli, Dino II-387
Manea, Florin II-289
Mann, Makai II-461
Mansur, Muhammad Numair II-777
Mariano, Benjamin II-777
Markgraf, Oliver II-243

Martin-Martin, Enrique II-863
Matheja, Christoph II-524
Mathews, N. II-851
McKinnis, Aaron I-566
Meel, Kuldeep S. II-313
Meier, Severin I-718
Merayo, Alicia II-863
Millstein, Todd II-577
Mitra, Sayan I-580
Mohan, Anshuman II-801
Mora, Federico II-289
Moser, Georg II-99
Mover, Sergio I-529
Müller, Peter I-367, I-718, II-704
Musau, Patrick I-263

Navas, Jorge A. I-201, II-777
Nicolet, Victor I-832
Niemetz, Aina II-231
Noller, Yannic I-3
Nowotka, Dirk II-289

Ölveczky, Peter Csaba I-491
Oortwijn, Wytse I-367
Osama, Muhammad II-447
Ouaknine, Joël II-172

Pal, Neelanjana I-263
Pappas, George I-249
Parthasarathy, Gaurav II-704
Păsăreanu, Corina S. I-3
Pastva, Samuel I-505
Pathak, Shreya I-341
Pavlogiannis, Andreas I-341
Peleg, Hila I-110
Pereira, João C. I-367
Pereira, Mário II-677
Perez, Mateo II-651
Petcher, Adam I-645
Peyras, Quentin II-337
Piterman, Nir I-767
Polikarpova, Nadia I-110
Prabhakar, Pavithra I-566
Pradella, Matteo II-387
Prakash, Karthik R. I-619
Preiner, Mathias II-231
Pulte, Christopher I-303
Purser, David II-172

Rain, Sophie I-317
Rakamarić, Zvonimir II-626
Ravara, António II-677
Reinhard, Tobias II-27
Reps, Thomas I-46, I-84, I-783
Rong, Dennis Liew Zhen I-403
Roşu, Grigore II-477
Roux, Cody I-808
Rowe, Reuben N. S. I-110
Roy, Diptarko II-3
Rubio, Albert II-863
Ryou, Wonryong I-225

Šafránek, David I-505
Sagiv, Mooly I-317
Sakr, Mouhammad II-435
Salvia, Rocco II-626
Sánchez, César I-694
Santos, José Fragoso II-827
Schewe, Sven II-651
Schröer, Philipp II-524
Sergey, Ilya I-110
Seshia, Sanjit A. II-553, II-577, II-602
Sewell, Peter I-303
Shi, Xiaomu II-149
Shoukry, Yasser I-287
Shriver, David I-137
Sibai, Hussein I-580
Siber, Julian I-894
Simner, Ben I-303
Singh, Gagandeep I-225
Singher, Eytan II-125
Slobodova, Anna I-26
Solar-Lezama, Armando I-808
Somenzi, Fabio II-651
Song, Fu I-175
Stan, Daniel II-243
Stefanescu, Andrei I-645
Strejček, Jan II-887
Stupinský, Šimon I-856
Summers, Alexander J. II-704
Sumners, Rob I-26
Sun, Youcheng I-3
Swords, Sol I-26

Tabajara, Lucas Martinelli I-870
Tang, Xiaochao I-467
Terauchi, Tachio I-742
Tkachuk, Oksana I-767
Toman, Viktor I-341

Tomovič, Lukáš II-887
Tonetta, Stefano I-529
Torfah, Hazem II-553
Tran, Hoang-Dung I-263
Tremblay, Hugo II-500
Trentin, P. II-851
Trinh, Minh-Thai II-477
Trivedi, Ashutosh II-651
Tsai, Ming-Hsien II-149

Unadkat, Divyesh II-911
Unno, Hiroshi I-742, II-75
Usman, Muhammad I-3

Vafeiadis, Viktor I-427
Van den Broeck, Guy II-577
van der Berg, Freark I. II-690
Vardi, Moshe Y. I-870
Vazquez-Chanlatte, Marcell II-577
Vechev, Martin I-225

Wahl, Thomas I-380
Wang, Bow-Yaw II-149
Wang, Qiuye I-443
Wang, Yuting II-728
Weimer, James I-249
Weiss, Gera I-870
Wijs, Anton II-447
Wojtczak, Dominik II-651
Wolf, Felix A. I-367
Worrell, James II-172
Wu, Jinhua II-728
Wüstholz, Valentin I-201, II-777

Xu, Xiangzhe II-728
Xue, Bai I-443

Yang, Bo-Yin II-149
Yang, Xiaodong I-263
Yang, Yahan II-461
Yang, Zhengfeng I-467
Yin, Zhenguo II-728
Ying, Mingsheng I-151
Yu, Emily II-363

Zeng, M. Q. II-851
Zeng, Xia I-467
Zeng, Zhenbing I-467
Zhan, Naijun I-443
Zhang, Hongce II-461
Zhang, Yedi I-175
Zhang, Yidan I-467
Zhang, Zhenya I-595
Zhao, Jianjun I-595
Zhao, Zhe I-175
Zhu, Shaowei II-51
Zicarelli, Hannah I-403
Zuleger, Florian II-99