Algorithm 4: A construction of $\mathsf{PATH} : S_{\Diamond \mathbf{1}} \to S^+$ for Lemma 4.8

1 $S_{\mathrm{v}} \leftarrow \{\mathbf{1}\}$, $\mathsf{PATH}(\mathbf{1}) \leftarrow \mathbf{1}$
2 **while** $S_{\Diamond \mathbf{1}} \setminus S_{\mathrm{v}} \neq \emptyset$ **do**
3 Choose a pair of states $(s_{\mathrm{c}}, s_{\mathrm{p}})$ that satisfies the following: $s_{\mathrm{c}} \in S \setminus S_{\mathrm{v}}$, $s_{\mathrm{p}} \in S_{\mathrm{v}}$, $V(\mathcal{G})(s_{\mathrm{c}}) = \max_{s \in S \setminus S_{\mathrm{v}}} V(\mathcal{G})(s)$, and for an optimal action a at s_{c} in $\mathcal{M}$, $s_{\mathrm{p}} \in \mathrm{post}(s_{\mathrm{c}}, a)$
4 $\mathsf{PATH}(s_{\mathrm{c}}) \leftarrow s_{\mathrm{c}} \cdot \mathsf{PATH}(s_{\mathrm{p}})$, $S_{\mathrm{v}} \leftarrow S_{\mathrm{v}} \cup \{s_{\mathrm{c}}\}$
5 **return** PATH

for this paid price (namely the information lost in the rough encoding) is that the resulting data structure (WG) allows fast *global* analysis via the widest path problem. Our experiment results in Sect. 5 demonstrate that this rough yet global approximation can make upper bounds quickly converge.

4.3 Soundness and Convergence

In Algorithm 3, an SG $\mathcal{G}$ is turned into an MDP $\mathcal{M}_i$ and then to a WG $\mathcal{W}_i$. Our claim is that computing a widest path in $\mathcal{W}_i$ gives the next upper bound U_i in the iteration. Here we prove the following correctness properties: soundness ($V(\mathcal{G}) \leq U_i$) and convergence ($U_i \to V(\mathcal{G})$ as $i \to \infty$).

We start with a technical lemma. The choice of the MDP $\mathcal{M}(\mathcal{G}, \mathrm{Av}')$ and the value function $V(\mathcal{G})$ (for $\mathcal{G}$, not for $\mathcal{M}(\mathcal{G}, \mathrm{Av}')$) in the statement is subtle; it turns out to be just what we need.

Lemma 4.8. *Let $\mathcal{G}$ be as in Algorithm 3, and $\mathrm{Av}' \colon S \to 2^A$ be a Minimizer restriction (Definition 4.2). Let $s_0 \in S_{\Diamond \mathbf{1}}$ be a state with a non-zero value (Definition 2.5). Consider the MDP $\mathcal{M}(\mathcal{G}, \mathrm{Av}')$ (Definition 4.1), for which we write simply $\mathcal{M}$. Then there is a finite path $\pi = s_0 a_0 s_1 a_1 \dots a_{n-1} s_n$ in $\mathcal{M}$ that satisfies the following.*

- *The path π reaches $\mathbf{1}$, that is, $s_n = \mathbf{1}$.*
- *Each action is optimal in $\mathcal{M}$ with respect to $V(\mathcal{G})$, that is, $\big(\mathbb{X}_{a_i}\big(V(\mathcal{G})\big)\big)(s_i) = \max_{a \in \mathrm{Av}'(s_i)} \big(\mathbb{X}_a\big(V(\mathcal{G})\big)\big)(s_i)$ for each $i \in [0, n-1]$.*
- *The value function $V(\mathcal{G})$ does not decrease along the path, that is, $V(\mathcal{G})(s_i) \leq V(\mathcal{G})(s_{i+1})$ for each $i \in [0, n-1]$.*

Proof. We construct a function $\mathsf{PATH} : S_{\Diamond \mathbf{1}} \to S^+$ by Algorithm 4. It is clear that PATH assigns a desired path to each $s_0 \in S_{\Diamond \mathbf{1}}$. In particular, $V(\mathcal{G})$ does not decrease along $\mathsf{PATH}(s_0)$ since always a state with a smaller value of $V(\mathcal{G})$ is prepended.

It remains to be shown that, in Line 3, a required pair $(s_{\mathrm{c}}, s_{\mathrm{p}})$ is always found. Let $S_{\mathrm{v}} \subsetneq S_{\Diamond \mathbf{1}}$ be a subset with $\mathbf{1} \in S_{\mathrm{v}}$; here S_{v} is a proper subset of $S_{\Diamond \mathbf{1}}$ since otherwise we should be already out of the while loop (Line 2).

Let $S_{\max} = \{s \in S \setminus S_{\mathrm{v}} \mid V(\mathcal{G})(s) = \max_{s' \in S \setminus S_{\mathrm{v}}} V(\mathcal{G})(s')\}$. Since $S_{\mathrm{v}} \subsetneq S_{\Diamond \mathbf{1}}$, we have $\emptyset \neq S_{\max} \subseteq S_{\Diamond \mathbf{1}}$ and thus $V(\mathcal{G})(s) > 0$ for each $s \in S_{\max}$. We also have $\mathbf{1} \notin S_{\max}$ since $\mathbf{1} \in S_{\mathrm{v}}$.

We argue by contradiction: assume that for any $s \in S \setminus S_{\mathrm{v}}$, $s' \in S_{\mathrm{v}}$, we have $s' \notin \mathrm{post}(s, a_s)$, where a_s is any optimal action at s in $\mathcal{M}$ with respect to $V(\mathcal{G})$.

Now let $s \in S_{\max}$ be an arbitrary element. It follows that $V(\mathcal{G})(s) > 0$.

$$
\begin{aligned}
V(\mathcal{G})(s) &\leq \left(\mathbb{X}_{a_s}(V(\mathcal{G}))\right)(s) \\
&\qquad \text{using Lemma 4.6; here } a_s \text{ is an optimal action at } s \text{ in } \mathcal{M} \text{ with respect to } V(\mathcal{G}), \\
&= \textstyle\sum_{s' \in S \setminus S_{\mathrm{v}}} \delta(s, a_s, s') \cdot V(\mathcal{G})(s') \\
&\qquad\qquad \text{by the assumption that } s' \notin \mathrm{post}(s, a_s) \text{ for each } s' \in S_{\mathrm{v}} \\
&\leq \textstyle\sum_{s' \in S \setminus S_{\mathrm{v}}} \delta(s, a_s, s') \cdot V(\mathcal{G})(s) \\
&\qquad\qquad \text{since } s \in S_{\max} \text{ and hence } V(\mathcal{G})(s') \leq V(\mathcal{G})(s) \\
&= V(\mathcal{G})(s) \qquad \text{since } \textstyle\sum_{s' \in S \setminus S_{\mathrm{v}}} \delta(s_{\mathrm{c}}, a, s') = 1. \qquad (10)
\end{aligned}
$$

Therefore both inequalities in the above must be equalities. In particular, for the second inequality (in (10)) to be an equality, we must have the weight for each suboptimal s' to be 0. That is, $\delta(s, a_s, s') = 0$ for each $s' \in (S \setminus S_{\mathrm{v}}) \setminus S_{\max}$.

The above holds for arbitrary $s \in S_{\max}$. Therefore, for any strategy that is optimal in $\mathcal{M}$ with respect to $V(\mathcal{G})$, once a play is in $S_{\max}$, it never comes out of $S_{\max}$, hence the play never reaches $\mathbf{1}$. Moreover, an optimal strategy in $\mathcal{M}$ with respect to $V(\mathcal{G})$ is at least as good as an optimal strategy for Maximizer in $\mathcal{G}$ (with respect to $V(\mathcal{G})$), that is, the latter reaches $\mathbf{1}$ no more often than the former. This follows from Lemma 4.6. Altogether, we conclude that a Maximizer optimal strategy in $\mathcal{G}$ does not lead any $s \in S_{\max}$ to $\mathbf{1}$, i.e., $V(\mathcal{M})(s) = 0$ for each $s \in S_{\max}$. Now we come to a contradiction. □

In the following lemma, we use the value function $V(\mathcal{G})$ in the position of f in Definition 4.7. This cannot be done in actual execution of Algorithm 4: unlike U_{i-1} in Algorithm 3, the value function $V(\mathcal{G})$ is not known to us. Nevertheless, the lemma is an important theoretical vehicle towards soundness of Algorithm 3.

Lemma 4.9. *Let $\mathcal{G}$ be the game in Algorithm 3, and $\mathrm{Av}' \colon S \to 2^A$ be a Minimizer restriction (Definition 4.2). Let $\mathcal{M} = \mathcal{M}(\mathcal{G}, \mathrm{Av}')$, and $\mathcal{W} = \mathcal{W}_{\mathrm{LcPg}}\big(\mathcal{M}, V(\mathcal{G})\big)$. Then, for each state $s \in S$, we have $\mathrm{WPW}(\mathcal{W})(s, \mathbf{1}) \geq V(\mathcal{G})(s)$.*

Proof. In what follows, we let the WG $\mathcal{W} = \mathcal{W}_{\mathrm{LcPg}}\big(\mathcal{M}, V(\mathcal{G})\big)$ be denoted by $\mathcal{W} = (S, E, w)$. Let $\pi = s_0 a_0 s_1 a_1 \dots a_{n-1} s_n$ be a path of the MDP $\mathcal{M}$ such that $s_n = \mathbf{1}$, each action is optimal in $\mathcal{M}$ with respect to $V(\mathcal{G})$, and $V(\mathcal{G})(s_i) \leq V(\mathcal{G})(s_{i+1})$ for each $i \in [0, n-1]$. Existence of such a path π is shown by Lemma 4.8. Let $\pi' = s_0 s_1 \dots s_{n-1} \mathbf{1}$ be the path in the WG $\mathcal{W}$ induced by π—we simply omit actions.

The path π' satisfies the following, for each $i \in [0, n-1]$.

$$
\begin{aligned}
w(s_i, s_{i+1}) &= \max\{\, (\mathbb{X}_a(V(\mathcal{G})))(s_i) \mid a \in \mathrm{Av}'(s_i), s_{i+1} \in \mathrm{post}(s_i, a) \,\} \quad \text{by Definition 4.7} \\
&= (\mathbb{X}_{a_i}(V(\mathcal{G})))(s_i) \quad \text{since } a_i \text{ is optimal wrt. } V(\mathcal{G}); \\
&\quad \text{note that } a_i \in \mathrm{Av}'(s_i), s_{i+1} \in \mathrm{post}(s_i, a_i) \text{ hold since } \pi \text{ is a path in } \mathcal{M} \\
&= \max_{a \in \mathrm{Av}'(s)} (\mathbb{X}_a(V(\mathcal{G})))(s_i) \quad \text{since } a_i \text{ is optimal wrt. } V(\mathcal{G}) \\
&\geq V(\mathcal{G})(s_i) \qquad \text{by Lemma 4.6.}
\end{aligned}
$$

This observation, combined with $V(\mathcal{G})(s_0) \leq V(\mathcal{G})(s_1) \leq \cdots \leq V(\mathcal{G})(s_n)$ (by the definition of π), implies that the width of the path π' is at least $V(\mathcal{G})(s_0)$. The widest path width is no smaller than that. □

Theorem 4.10 (soundness). *In Algorithm 3, $V(\mathcal{G}) \leq U_i$ holds for each $i \in \mathbb{N}$.*

Proof. We let the function

$$
\begin{aligned}
&\min\{\, U,\ \mathrm{WPW}(\mathcal{W}_{\mathrm{LcPg}}(\mathcal{M}(\mathcal{G}, \mathrm{Av}'), U))(\_, \mathbf{1}) \,\} \quad : \quad S \longrightarrow [0,1] \\
&\text{denoted by} \qquad T(\mathrm{Av}', U) : \quad S \longrightarrow [0,1],
\end{aligned}
$$

clarifying its dependence on Av' and $U \colon S \to [0,1]$. Clearly, for each $i \in \mathbb{N}$, we have $U_i = T(\mathrm{Av}_{L_i}, U_{i-1})$.

The rest of the proof is by induction. It is trivial if $i = 0$ ($U_0 = \top$).

$$
\begin{aligned}
U_{i+1} &= T(\mathrm{Av}_{L_i}, U_i) \\
&\geq T(\mathrm{Av}_{L_i}, V(\mathcal{G})) \qquad \text{by ind. hyp., and } T(\mathrm{Av}_{L_i}, \_) \text{ is monotone} \\
&= \min\{\, V(\mathcal{G}),\ \mathrm{WPW}(\mathcal{W}_{\mathrm{LcPg}}(\mathcal{M}(\mathcal{G}, \mathrm{Av}_{L_i}), V(\mathcal{G})))(\_, \mathbf{1}) \,\} \\
&= V(\mathcal{G}) \qquad \text{by Lemma 4.9.}
\end{aligned}
$$

□

It is clear that U_i decreases with respect to i ($U_0 \geq U_1 \geq \cdots$), by the presence of min in Line 8. It remains to show the following.

Theorem 4.11 (convergence). *In Algorithm 3, let the while loop iterate forever. Then $U_i \to V(\mathcal{G})$ as $i \to \infty$.*

Proof. We give a proof using the infinitary pigeonhole principle. The proof is nonconstructive—it is not suited for analyzing the speed of convergence, for example—but the proof becomes simpler.

In what follows, we let $\mathbb{X}_\sigma \colon (S \to [0,1]) \to (S \to [0,1])$ denote the Bellman operator on an MDP $\mathcal{M}$ induced by a strategy σ, i.e., $(\mathbb{X}_\sigma f)(s) := (\mathbb{X}_{\sigma(s)} f)(s)$. The MC obtained from an MDP $\mathcal{M}$ by fixing a strategy σ is denoted by $\mathcal{M}^\sigma$.

Towards the statement of the theorem, for each $i \in \mathbb{N}$, we choose a (positional) strategy σ_i in the MDP $\mathcal{M}_i$ as follows.

– For each $s \in S_{\Diamond\mathbf{1}}$, take the widest path $\mathrm{WPath}(\mathcal{W}_i, \mathbf{1})(s) = ss_1 \dots \mathbf{1}$ in $\mathcal{W}_i$ from s to $\mathbf{1}$ (Definition 2.8). Such a path from s to $\mathbf{1}$ exists—otherwise we have $U_i(s) = 0$, hence $V(\mathcal{G})(s) = 0$ by Theorem 4.10.
Let $\sigma_i(s)$ be an action that justifies the first edge in the chosen widest path, that is, $a \in \mathrm{Av}_i(s)$ such that $s_1 \in \mathrm{post}(s, a)$.
– For each $s \in S \setminus S_{\Diamond\mathbf{1}}$, $\sigma_i(s)$ is freely chosen from $\mathrm{Av}_i(s)$.

It is then easy to see that

$$\mathrm{WPW}(\mathcal{W}_i)(s) \le (\mathbb{X}_{\sigma_i} U_{i-1})(s) \qquad \text{for each } i \in \mathbb{N} \text{ and } s \in S_{\Diamond\mathbf{1}}. \tag{11}$$

Indeed, by the definition of σ_i, the right-hand side is the weight of the first edge in the chosen widest path. This must be no smaller than the widest path width, that is, the width of the chosen path.

Now, since there are only finitely many strategies for the SG $\mathcal{G}$, the same is true for the MDPs $\mathcal{M}_0, \mathcal{M}_1, \dots$ that are obtained from $\mathcal{G}$ by restricting Minimizer's actions. Therefore, by the infinitary pigeonhole principle, there are infinitely many $i_0 < i_1 < \cdots$ such that $\sigma_{i_0} = \sigma_{i_1} = \cdots \quad =: \sigma^\dagger$. Moreover, we can choose them so that they are all beyond i_{M} in Lemma 4.5, in which case we have

$$V(\mathcal{M}_{i_m}^{\sigma^\dagger}) \le V(\mathcal{G}) \quad \text{for each } m \in \mathbb{N}. \tag{12}$$

Indeed, Minimizer's actions are already optimized in $\mathcal{M}_i$ (Lemma 4.5), and thus the only freedom left for $\sigma^\dagger$ is to choose suboptimal actions of Maximizer's.

In what follows, we cut down the domain of discourse from $S \to [0,1]$ to $S_{\Diamond\mathbf{1}} \to [0,1]$, i.e., 1) every function of the type $f : S \to [0,1]$ is now seen as the restriction over $S_{\Diamond\mathbf{1}}$, and 2) the Bellman operator only adds up the value of the input function over $S_{\Diamond\mathbf{1}}$, namely it is now defined by $\hat{\mathbb{X}}_a f(s) = \sum_{s' \in S_{\Diamond\mathbf{1}}} \delta(s, a, s') \cdot f(s')$. The operator $\hat{\mathbb{X}}_\sigma$ is also defined in a similar way to $\mathbb{X}_\sigma$.

Now proving convergence in $S_{\Diamond\mathbf{1}} \to [0,1]$ suffices for the theorem. Indeed, for each $i \ge i_{\mathrm{M}}$, we have $V(\mathcal{M}_i)(s) = V(\mathcal{G})(s) = 0$ for each $s \in S \setminus S_{\Diamond\mathbf{1}}$. This implies that there is no path from s to $\mathbf{1}$ in $\mathcal{M}_i$, thus neither in the WG $\mathcal{W}_i$. Therefore $U_i \le \mathrm{WPW}(\mathcal{W}_i) = 0$.

A benefit of this domain restriction is that the Bellman operator $\hat{\mathbb{X}}_\sigma$ has a unique fixed point in $S_{\Diamond\mathbf{1}} \to [0,1]$ if the set of non-sink states in $\mathcal{M}^\sigma$ is exactly $S_{\Diamond\mathbf{1}}$, i.e., $V(\mathcal{M}^\sigma)(s) > 0$ holds if and only if $s \in S_{\Diamond\mathbf{1}}$. Furthermore, this unique fixed point is the value function $V(\mathcal{M}^\sigma)$ restricted to $S_{\Diamond\mathbf{1}} \subseteq S$ [4, Theorem 10.19]. Therefore $V(\mathcal{M}^\sigma)$ is computed by the gfp Kleene iteration, too:

$$\top \ge \hat{\mathbb{X}}_\sigma \top \ge (\hat{\mathbb{X}}_\sigma)^2 \top \ge \cdots \longrightarrow V(\mathcal{M}^\sigma) \quad \text{in the space } S_{\Diamond\mathbf{1}} \to [0,1]. \tag{13}$$

We show the following by induction on m.

$$U_{i_m} \le (\hat{\mathbb{X}}_{\sigma^\dagger})^m \top \quad \text{for each } m \in \mathbb{N}. \tag{14}$$

It is obvious for $m = 0$. For the step case, we have the following. Notice that the inequality (11) holds in the restricted domain for $i \geq i_M$.

$$\begin{aligned} U_{i_{m+1}} &\leq \mathrm{WPW}(\mathcal{W}_{i_{m+1}}) \quad \text{by Line 8 of Algorithm 3} \\ &\leq \hat{\mathbb{X}}_{\sigma^\dagger} U_{i_{m+1}-1} \quad \text{by (11)} \\ &\leq \hat{\mathbb{X}}_{\sigma^\dagger} U_{i_m} \quad \text{by monotonicity of } \hat{\mathbb{X}}_{\sigma^\dagger}, \text{ decrease of } U_i \text{ and } i_m < i_{m+1} \\ &\leq (\hat{\mathbb{X}}_{\sigma^\dagger})^{m+1} \top \quad \text{by the induction hypothesis.} \end{aligned}$$

We have proved (14) which proves $\inf_i U_i \leq \inf_m (\hat{\mathbb{X}}_{\sigma^\dagger})^m \top$.

Lastly, we prove that $V(\mathcal{M}_{i_m}^{\sigma^\dagger})(s) > 0$ holds if and only if $s \in S_{\Diamond \mathbf{1}}$ for each $m \in \mathbb{N}$, and thus $\sigma^\dagger$ follows the characterization in (13). This proves

$$\inf_i U_i \leq V(\mathcal{M}_{i_m}^{\sigma^\dagger}) \quad \text{for each } m \in \mathbb{N}. \tag{15}$$

Implication to the right is clear as Minimizer restriction is done optimally in $\mathcal{M}_{i_m}$. Conversely, if $s \in S_{\Diamond \mathbf{1}}$, then there is a path from s to $\mathbf{1}$ in $\mathcal{W}_{i_m}$. Let $\mathrm{WPath}(\mathcal{W}_{i_m}, \mathbf{1})(s) = s_0 s_1 \dots s_k$, where $s_0 = s$, $k \in \mathbb{N}$ and $s_k = \mathbf{1}$. Then by the property of WPath and $\sigma^\dagger$, we have $\delta(s_j, \sigma^\dagger(s_j), s_{j+1}) > 0$ for each $j < k$. Thus, the probability that the finite path $\mathrm{WPath}(\mathcal{W}_{i_m}, \mathbf{1})(s)$ is obtained by running $\mathcal{M}_{i_m}^{\sigma^\dagger}$ starting from s, which is apparently at most $V(\mathcal{M}_{i_m}^{\sigma^\dagger})(s)$, is nonzero. Hence we have implication to the left.

Combining (12), (15) and Theorem 4.10, we obtain the claim. □

5 Experiment Results

Experiment Settings. We compare the following four algorithms.

- *WP* is our BVI algorithm via widest paths. It avoids end component (EC) computation by global propagation of upper bounds.
- *DFL* is the implementation of the main algorithm in [20]. It relies on EC computation for deflating.
- *DFL_m* is our modification of DFL, where some unnecessary repetition of EC computation is removed.
- *DFL_BRTDP* is the learning-based variant of DFL. It restricts bound update to those states which are visited by simulations. See [20] for details.

The latter three—coming from [20]—are the only existing BVI algorithms for SGs with a convergence guarantee, to the best of our knowledge. The implementation of DFL and DFL_BRTDP is provided by the authors of [20].

The four algorithms are implemented on top of PRISM-games [21] version 2.0. We used the stopping threshold $\varepsilon = 10^{-6}$. The experiments were conducted on Dell Inspiron 3421 Laptop with 4.00 GB RAM and Intel(R) Core(TM) i5-3337U 1.80 GHz processor.

In the implementations of DFL and DFL_BRTDP, the deflating operation is applied only once every five iterations [20, Sect. B.3]. Following this, our WP also

solves the widest path problem (Line 8) only once every five iterations, while other operations are applied in each iteration.

For input SGs, we took four models from the literature: *mdsm* [11], *cloud* [6], *teamform* [12] and *investor* [22]. In addition, we used our model *manyECs*—an artificial model with many ECs—to assess the effect of ECs on performance. The model manyECs is presented in the appendix in [24]. Each of these five models comes with a model parameter N.

There is another model called *cdmsn* in [20]. We do not discuss cdmsn since all the algorithms (ours and those from [20]) terminated within 0.001 seconds.

Results. The number i of iterations and the running time for each algorithm and each input SG is shown in Table 1. For DFL_BRTDP, the ratio of states visited by the algorithm is shown in percentage; the smaller it is, the more efficient the algorithm is in reducing the state space. Each number for DFL_BRTDP (a probabilistic algorithm) is the average over 5 runs.

Table 1. Experimental results, comparing WP (our algorithm) with those in [20]. N is a model parameter (the bigger the more complex). #states, #trans, #EC show the numbers of states, transitions and ECs in the SG, respectively. itr is the number i of iterations at termination; time is the execution time in seconds. For each SG, the fastest algorithm is shaded in green. The settings that did not terminate are shaded in gray; TO is time out (6 h), OOM is out of memory, and SO is stack overflow.

model	N	#states	#trans	#EC	DFL		DFL_m		DFL_BRTDP			WP	
					itr	time	itr	time	itr	visit%	time	itr	time
mdsm	3	62245	151143	1	121	3	121	4	17339	49.3	15	120	5
	4	335211	882765	1	125	15	125	47	91301	42.1	86	124	38
cloud	5	8842	60437	4421	7	7	7	1	167	6.9	14	7	<1
	6	34954	274965	17477	11	177	11	5	41	0.6	3	11	1
	7	139402	1237525	69701	11	19721	11	62	41	0.2	4	11	5
teamform	3	12475	15228	2754	2	<1	2	<1	972	49.0	137	2	<1
	4	96665	116464	19800	2	<1	2	<1	4154	34.6	9603	2	<1
	5	907993	1084752	176760	2	<1	2	<1			TO	2	<1
investor	50	211321	673810	29690	441	184	441	249			TO	364	48
	100	807521	2587510	114390	801	3318		OOM			TO	688	736
manyECs	500	1004	3007	502	6	7	6	7			TO	5	<1
	1000	2004	6007	1002	6	51	6	51			TO	5	<1
	5000	10004	30007	5002		SO		SO			TO	5	<1

Discussion. We observe consistent performance advantage of our algorithm (WP). Even in the mdsm model where the DFL algorithms do not suffer from EC computation (#EC is just 1), WP's performance is comparable to DFL. The cloud model is where the learning-based approach in [20] works well—see visit% that are very small. Our WP performs comparably against DFL_BRTDP, too.

The performance advantage of our WP algorithm is eminent, not only in the artificial model of manyECs (where WP is faster by magnitudes), but also in

the realistic model investor that comes from a financial application scenario [22]. The results for these two models suggest that WP is indeed advantageous when EC computation poses a bottleneck for other algorithms.

Overall, we observe that our WP algorithm can be the first choice when it comes to solving SGs: for some models, it runs much faster than other algorithms; for other models, even if the performances of other algorithms differs a lot, WP's performance is comparable with the best algorithm.

6 Conclusions and Future Work

In this paper, we presented a new BVI algorithm for solving stochastic games. It features global propagation of upper bounds by widest paths, via a novel encoding of the problem to a suitable weighted graph. This way we avoid computation of end components that often penalizes the performance of the other BVI-based algorithms. Our experimental comparison with known BVI algorithms for SGs demonstrates the efficiency of our algorithm. For correctness of the algorithm, we presented proofs for soundness and convergence.

Extending the current algorithm for more advanced settings is future work—this is much like the results in [20] are extended and used in [2,3,16]. In doing so, we hope to make essential use of structures that are unique to those advanced problem settings. Another important direction is to push forward the idea of global propagation in verification and synthesis, seeking further instances of the idea. Finally, pursuing the global propagation idea in the context of reinforcement learning—where problems are often formalized using MDPs and the Bellman operator is heavily utilized—may open up another fruitful collaboration between formal methods and statistical machine learning.

Acknowledgment. The authors are supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST; I.H. is supported by Grant-in-Aid No. 15KT0012, JSPS. Thanks are due to Maximilian Weininger and Edon Kelmendi for sharing their implementation, and to Pranav Ashok and David Sprunger for useful discussions and comments.

References

1. Andersson, D., Miltersen, P.B.: The complexity of solving stochastic games on graphs. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 112–121. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10631-6_13
2. Ashok, P., Kretinsky, J., Weininger, M.: Approximating values of generalized-reachability stochastic games. CoRR abs/1908.05106 (2019). http://arxiv.org/abs/1908.05106
3. Ashok, P., Křetínský, J., Weininger, M.: PAC statistical model checking for Markov decision processes and stochastic games. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 497–519. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_29

4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. 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
6. Calinescu, R., Kikuchi, S., Johnson, K.: Compositional reverification of probabilistic safety properties for large-scale complex IT systems. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 303–329. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_16
7. Chatterjee, K., Dvorák, W., Henzinger, M., Svozil, A.: Near-linear time algorithms for streett objectives in graphs and MDPS. In: Fokkink, W., van Glabbeek, R. (eds.) 30th International Conference on Concurrency Theory CONCUR 2019, 27–30 August 2019, Amsterdam, the Netherlands. LIPIcs, vol. 140, pp. 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.CONCUR.2019.7
8. Chatterjee, K., Fijalkow, N.: A reduction from parity games to simple stochastic games. In: D'Agostino, G., La Torre, S. (eds.) Proceedings of Second International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2011, Minori, Italy, 15–17 June 2011. EPTCS, vol. 54, pp. 74–86 (2011). https://doi.org/10.4204/EPTCS.54.6
9. Chatterjee, K., Henzinger, M.: Efficient and dynamic algorithms for alternating büchi games and maximal end-component decomposition. J. ACM (JACM) **61**(3), 15 (2014)
10. Chatterjee, K., Henzinger, T.A.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_7
11. Chen, T., Forejt, V., Kwiatkowska, M.Z., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Formal Methods Syst. Design **43**(1), 61–92 (2013). https://doi.org/10.1007/s10703-013-0183-7
12. Chen, T., Kwiatkowska, M., Parker, D., Simaitis, A.: Verifying team formation protocols with probabilistic model checking. In: Leite, J., Torroni, P., Ågotnes, T., Boella, G., van der Torre, L. (eds.) CLIMA 2011. LNCS (LNAI), vol. 6814, pp. 190–207. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22359-4_14
13. Condon, A.: The complexity of stochastic games. Inf. Comput. **96**(2), 203–224 (1992). https://doi.org/10.1016/0890-5401(92)90048-K
14. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995). https://doi.org/10.1145/210332.210339
15. De Alfaro, L.: Formal verification of probabilistic systems. Citeseer (1997)
16. Eisentraut, J., Kretinsky, J., Rotar, A.: Stopping criteria for value and strategy iteration on concurrent stochastic reachability games. CoRR abs/1909.08348 (2019). http://arxiv.org/abs/1909.08348
17. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987). https://doi.org/10.1145/28869.28874
18. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theoret. Comput. Sci. **735**, 111–131 (2018)
19. Hoffman, A.J., Karp, R.M.: On nonterminating stochastic games. Manage. Sci. **12**(5), 359–370 (1966). https://doi.org/10.1287/mnsc.12.5.359

20. Kelmendi, E., Krämer, J., Křetínský, J., Weininger, M.: Value iteration for simple stochastic games: stopping criterion and learning algorithm. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 623–642. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_36
21. Kwiatkowska, M., Parker, D., Wiltsche, C.: PRISM-games: verification and strategy synthesis for stochastic multi-player games with multiple objectives. Int. J. Softw. Tools Technol. Transf. **20**(2), 195–210 (2017)
22. McIver, A., Morgan, C.: Results on the quantitative μ-calculus qmμ. ACM Trans. Comput. Log. **8**(1), 3 (2007). https://doi.org/10.1145/1182613.1182616
23. McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: Raedt, L.D., Wrobel, S. (eds.) Machine Learning, Proceedings of the Twenty-Second International Conference (ICML 2005), Bonn, Germany, 7–11 August 2005. ACM International Conference Proceeding Series, vol. 119, pp. 569–576. ACM (2005). https://doi.org/10.1145/1102351.1102423
24. Phalakarn, K., Takisaka, T., Haas, T., Hasuo, I.: Widest paths and global propagation in bounded value iteration for stochastic games. arXiv preprint (2020)
25. Svorenová, M., Kwiatkowska, M.: Quantitative verification and strategy synthesis for stochastic games. Eur. J. Control **30**, 15–30 (2016). https://doi.org/10.1016/j.ejcon.2016.04.009
26. Ujma, M.: On Verication and Controller Synthesis for Probabilistic Systems at Runtime. Ph.D. thesis, Wolfson College, University of Oxford (2015)

Checking Qualitative Liveness Properties of Replicated Systems with Stochastic Scheduling

Michael Blondin[1], Javier Esparza[2], Martin Helfrich[2], Antonín Kučera[3], and Philipp J. Meyer[2(✉)]

[1] Université de Sherbrooke, Sherbrooke, Canada
michael.blondin@usherbrooke.ca
[2] Technical University of Munich, Munich, Germany
{esparza,helfrich,meyerphi}@in.tum.de
[3] Masaryk University, Brno, Czechia
tony@fi.muni.cz

Abstract. We present a sound and complete method for the verification of qualitative liveness properties of replicated systems under stochastic scheduling. These are systems consisting of a finite-state program, executed by an unknown number of indistinguishable agents, where the next agent to make a move is determined by the result of a random experiment. We show that if a property of such a system holds, then there is always a witness in the shape of a *Presburger stage graph*: a finite graph whose nodes are Presburger-definable sets of configurations. Due to the high complexity of the verification problem (non-elementary), we introduce an incomplete procedure for the construction of Presburger stage graphs, and implement it on top of an SMT solver. The procedure makes extensive use of the theory of well-quasi-orders, and of the structural theory of Petri nets and vector addition systems. We apply our results to a set of benchmarks, in particular to a large collection of population protocols, a model of distributed computation extensively studied by the distributed computing community.

Keywords: Parameterized verification · Liveness · Stochastic systems

1 Introduction

Replicated systems consist of a fully symmetric finite-state program executed by an unknown number of indistinguishable agents, communicating by rendez-vous

Michael Blondin is supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Fonds de recherche du Québec – Nature et technologies (FRQNT). Javier Esparza, Martin Helfrich and Philipp J. Meyer have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367 (PaVeS). Antonín Kučera is supported by the Czech Science Foundation, grant No. 18-11193S.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 372–397, 2020.
https://doi.org/10.1007/978-3-030-53291-8_20

or via shared variables [14,16,41,46]. Examples include distributed protocols and multithreaded programs, or abstractions thereof. The communication graph of replicated systems is a clique. They are a special class of *parameterized systems*, i.e., infinite families of systems that admit a finite description in some suitable modeling language. In the case of replicated systems, the (only) parameter is the number of agents executing the program.

Verifying a replicated system amounts to proving that an infinite family of systems satisfies a given property. This is already a formidable challenge, made even harder by the fact that we want to verify liveness (more difficult than safety) against stochastic schedulers. Loosely speaking, stochastic schedulers select the set of agents that should execute the next action as the result of a random experiment. Stochastic scheduling often appears in distributed protocols, and in particular also in population protocols—a model much studied in distributed computing with applications in computational biology[1]—that supplies many of our case studies [9,58]. Under stochastic scheduling, the semantics of a replicated system is an infinite family of finite-state Markov chains. In this work, we study *qualitative* liveness properties, stating that the infinite runs starting at configurations of the system satisfying a precondition almost surely reach and stay in configurations satisfying a postcondition. In this case, whether the property holds or not depends only on the topology of the Markov chains, and not on the concrete probabilities.

We introduce a formal model of replicated systems, based on multiset rewriting, where processes can communicate by shared variables or multiway synchronization. We present a sound and complete verification method called *Presburger stage graphs*. A Presburger stage graphs is a directed acyclic graphs with Presburger formulas as nodes. A formula represents a possibly infinite inductive set of configurations, i.e., a set of configurations closed under reachability. A node $\mathcal{S}$ (which we identify with the set of configurations it represents) has the following property: A run starting at any configuration of $\mathcal{S}$ almost surely reaches some configuration of some successor $\mathcal{S}'$ of $\mathcal{S}$, and, since $\mathcal{S}'$ is inductive, get trapped in $\mathcal{S}'$. A stage graph labels the node $\mathcal{S}$ with a witness of this property in the form of a *Presburger certificate*, a sort of ranking function expressible in Presburger arithmetic. The completeness of the technique, i.e., the fact that for every property of the replicated system that holds there exists a stage graph proving it, follows from deep results of the theory of vector addition systems (VASs) [52–54].

Unfortunately, the theory of VASs also shows that, while the verification problems we consider are decidable, they have non-elementary computational complexity [33]. As a consequence, verification techniques that systematically explore the space of possible stage graphs for a given property are bound to be very inefficient. For this reason, we design an incomplete but efficient algorithm for the computation of stage graphs. Inspired by theoretical results, the algorithm combines a solver for linear constraints with some elements of the theory of well-structured systems [2,39]. We report on the performance of this algorithm for a large number of case studies. In particular, the algorithm automatically verifies

[1] Under the name of *chemical reaction networks*.

many standard population protocols described in the literature [5,8,20,22,23,28,31], as well as liveness properties of distributed algorithms for leader election and mutual exclusion [3,40,42,44,50,59,61,64].

Related Work. The parameterized verification of replicated systems was first studied in [41], where they were modeled as counter systems. This allows one to apply many efficient techniques [11,24,37,47]. Most of these works are inherently designed for safety properties, and some can also handle fair termination [38], but none of them handles stochastic scheduling. To the best of our knowledge, the only works studying parameterized verification of liveness properties under our notion of stochastic scheduling are those on verification of population protocols. For *fixed* populations, protocols can be verified with standard probabilistic model checking [13,65], and early works follow this approach [28,31,60,63]. Subsequently, an algorithm and a tool for the *parameterized* verification of population protocols were described in [21,22], and a first version of stage graphs was introduced in [23] for analyzing the expected termination time of population protocols. In this paper we overhaul the framework of [23] for liveness verification, drawing inspiration from the safety verification technology of [21,22]. Compared to [21,22], our approach is not limited to a specific subclass of protocols, and captures models beyond population protocols. Furthermore, our new techniques for computing Presburger certificates subsume the procedure of [22]. In comparison to [23], we provide the first completeness and complexity results for stage graphs. Further, our stage graphs can prove correctness of population protocols and even more general liveness properties, while those of [23] can only prove termination. We also introduce novel techniques for computing stage graphs, which compared to [23] can greatly reduce their size and allows us to prove more examples correct.

There is also a large body of work on parameterized verification via cut-off techniques: one shows that a specification holds for any number of agents iff it holds for any number of agents below some threshold called the cutoff (see [6,26,30,34,46], and [16] for a comprehensive survey). Cut-off techniques can be applied to systems with an array or ring communication structure, but they require the existence and effectiveness of a cutoff, which is not the case in our setting. Further parameterized verification techniques are regular model checking [1,25] and automata learning [7]. The classes of communication structures they can handle are orthogonal to ours: arrays and rings for regular model checking and automata learning, and cliques in our work. Regular model checking and learning have recently been employed to verify safety properties [29], liveness properties under arbitrary schedulers [55] and termination under finitary fairness [51]. The classes of schedulers considered in [51,55] are incomparable to ours: arbitrary schedulers in [55], and finitary-fair schedulers in [51]. Further, these works are based on symbolic state-space exploration, while our techniques are based on automatic construction of invariants and ranking functions [16].

2 Preliminaries

Let $\mathbb{N}$ denote $\{0, 1, \ldots\}$ and let E be a finite set. A *unordered vector* over E is a mapping $V : E \to \mathbb{Z}$. In particular, a *multiset* over E is an unordered vector $M : E \to \mathbb{N}$ where $M(e)$ denotes the number of occurrences of e in M. The sets of all unordered vectors and multisets over E are respectively denoted $\mathbb{Z}^E$ and $\mathbb{N}^E$. Vector addition, subtraction and comparison are defined componentwise. The *size* of a multiset M is denoted $|M| = \sum_{e \in E} M(e)$. We let $E^{\langle k \rangle}$ denote the set of all multisets over E of size k. We sometimes describe multisets using a set-like notation, e.g.$M = \{\!|f, g, g|\!\}$ or equivalently $M = \{\!|f, 2 \cdot g|\!\}$ is such that $M(f) = 1$, $M(g) = 2$ and $M(e) = 0$ for all $e \notin \{f, g\}$.

Presburger Arithmetic. Let X be a set of variables. The set of formulas of *Presburger arithmetic* over X is the result of closing atomic formulas, as defined in the next sentence, under Boolean operations and first-order existential quantification. Atomic formulas are of the form $\sum_{i=1}^{k} a_i x_i \sim b$, where a_i and b are integers, x_i are variables and $\sim$ is either $<$ or $\equiv_m$, the latter denoting the congruence modulo m for any $m \geq 2$. Formulas over X are interpreted on $\mathbb{N}^X$. Given a formula ϕ of Presburger arithmetic, we let $[\![\phi]\!]$ denote the set of all multisets satisfying ϕ. A set $E \subseteq \mathbb{N}^X$ is a *Presburger set* if $E = [\![\phi]\!]$ for some formula ϕ.

2.1 Replicated Systems

A *replicated system* over Q of arity n is a tuple $\mathcal{P} = (Q, T)$, where $T \subseteq \bigcup_{k=0}^{n} Q^{\langle k \rangle} \times Q^{\langle k \rangle}$ is a *transition relation* containing the set of *silent* transitions $\bigcup_{k=0}^{n} \{(\boldsymbol{x}, \boldsymbol{x}) \mid \boldsymbol{x} \in Q^{\langle k \rangle})\}$[2]. A *configuration* is a multiset C of states, which we interpret as a global state with $C(q)$ agents in each state $q \in Q$.

For every $t = (\boldsymbol{x}, \boldsymbol{y}) \in T$ with $\boldsymbol{x} = \{\!|X_1, X_2, \ldots, X_k|\!\}$ and $\boldsymbol{y} = \{\!|Y_1, Y_2, \ldots, Y_k|\!\}$, we write $X_1 X_2 \cdots X_k \mapsto Y_1 Y_2 \cdots Y_k$ and let ${}^\bullet t \stackrel{\text{def}}{=} \boldsymbol{x}$, $t^\bullet \stackrel{\text{def}}{=} \boldsymbol{y}$ and $\Delta(t) \stackrel{\text{def}}{=} t^\bullet - {}^\bullet t$. A transition t is *enabled* at a configuration C if $C \geq {}^\bullet t$ and, if so, can *occur*, leading to the configuration $C' = C + \Delta(t)$. If t is not enabled at C, then we say that it is *disabled*. We use the following reachability notation:

$$
\begin{aligned}
&C \xrightarrow{t} C' \iff t \text{ is enabled at } C \text{ and its occurrence leads to } C',\\
&C \longrightarrow C' \iff C \xrightarrow{t} C' \text{ for some } t \in T,\\
&C \xrightarrow{w} C' \iff C = C_0 \xrightarrow{w_1} C_1 \cdots \xrightarrow{w_n} C_n = C' \text{ for some } C_0, C_1, \ldots, C_n \in \mathbb{N}^Q,\\
&C \xrightarrow{*} C' \iff C \xrightarrow{w} C' \text{ for some } w \in T^*.
\end{aligned}
$$

Observe that, by definition of transitions, $C \to C'$ implies $|C| = |C'|$, and likewise for $C \xrightarrow{*} C'$. Intuitively, transitions cannot create or destroy agents.

A *run* is an infinite sequence $C_0 t_1 C_1 t_2 C_2 \cdots$ such that $C_i \xrightarrow{t_{i+1}} C_{i+1}$ for every $i \geq 0$. Given $L \subseteq T^*$ and a set of configurations $\mathcal{C}$, we let

$$
post_L(\mathcal{C}) \stackrel{\text{def}}{=} \{C' : C \in \mathcal{C}, w \in L, C \xrightarrow{w} C'\}, \qquad post^*(\mathcal{C}) \stackrel{\text{def}}{=} post_{T^*}(\mathcal{C}),
$$

$$
pre_L(\mathcal{C}) \stackrel{\text{def}}{=} \{C : C' \in \mathcal{C}, w \in L, C \xrightarrow{w} C'\}, \qquad pre^*(\mathcal{C}) \stackrel{\text{def}}{=} pre_{T^*}(\mathcal{C}).
$$

[2] In the paper, we will omit the silent transitions when giving replicated systems.

Stochastic Scheduling. We assume that, given a configuration C, a probabilistic scheduler picks one of the transitions enabled at C. We only make the following two assumptions about the random experiment determining the transition: first, the probability of a transition depends only on C, and, second, every transition enabled at C has a nonzero probability of occurring. Since $C \xrightarrow{*} C'$ implies $|C| = |C'|$, the number of configurations reachable from any configuration C is finite. Thus, for every configuration C, the semantics of $\mathcal{P}$ from C is a finite-state Markov chain rooted at C.

Example 1. Consider the replicated system $\mathcal{P} = (Q, T)$ of arity 2 with states $Q = \{\mathrm{A_Y}, \mathrm{A_N}, \mathrm{P_Y}, \mathrm{P_N}\}$ and transitions $T = \{t_1, t_2, t_3, t_4\}$, where

$$t_1 \colon \mathrm{A_Y\,A_N} \mapsto \mathrm{P_Y\,P_N}, \qquad t_2 \colon \mathrm{A_Y\,P_N} \mapsto \mathrm{A_Y\,P_Y},$$
$$t_3 \colon \mathrm{A_N\,P_Y} \mapsto \mathrm{A_N\,P_N}, \qquad t_4 \colon \mathrm{P_Y\,P_N} \mapsto \mathrm{P_N\,P_N}.$$

Intuitively, at every moment in time, agents are either *Active* or *Passive*, and have output *Yes* or *No*, which corresponds to the four states of Q. This system is designed to satisfy the following property: for every configuration C in which all agents are initially active, i.e., C satisfies $C(\mathrm{P_Y}) = C(\mathrm{P_N}) = 0$, if $C(\mathrm{A_Y}) > C(\mathrm{A_N})$, then eventually all agents stay forever in the "yes" states $\{\mathrm{A_Y}, \mathrm{P_Y}\}$, and otherwise all agents eventually stay forever in the "no" states $\{\mathrm{A_N}, \mathrm{P_N}\}$. ◁

2.2 Qualitative Model Checking

Let us fix a replicated system $\mathcal{P} = (Q, T)$. Formulas of *linear temporal logic (LTL)* on $\mathcal{P}$ are defined by the following grammar:

$$\varphi ::= \phi \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi\,\mathbf{U}\,\varphi$$

where ϕ is a Presburger formula over Q. We look at ϕ as an atomic proposition over the set $\mathbb{N}^Q$ of configurations. Formulas of LTL are interpreted over runs of $\mathcal{P}$ in the standard way. We abbreviate $\Diamond\varphi \equiv \mathit{true}\,\mathbf{U}\,\varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$.

Let us now introduce the probabilistic interpretation of LTL. A configuration C of $\mathcal{P}$ satisfies an LTL formula φ *with probability* p if $\Pr[C, \varphi] = p$, where $\Pr[C, \varphi]$ denotes the probability of the set of runs of $\mathcal{P}$ starting at C that satisfy φ in the finite-state Markov chain rooted at C. The measurability of this set of runs for every C and φ follows from well-known results [65]. The *qualitative model checking problem* consists of, given an LTL formula φ and a set of configurations $\mathcal{I}$, deciding whether $\Pr[C, \varphi] = 1$ for every $C \in \mathcal{I}$. We will often work with the complement problem, i.e., deciding whether $\Pr[C, \neg\varphi] > 0$ for some $C \in \mathcal{I}$.

In contrast to the action-based qualitative model checking problem of [35], our version of the problem is undecidable due to adding atomic propositions over configurations (see the full version of the paper [19] for a proof):

Theorem 1. *The qualitative model checking problem is not semi-decidable.*

It is known that qualitative model checking problems of finite-state probabilistic systems reduces to model checking of non-probabilistic systems under an adequate notion of fairness.

Definition 1. *A run of a replicated system $\mathcal{P}$ is* fair *if for every possible step $C \xrightarrow{t} C'$ of $\mathcal{P}$ the following holds: if the run contains infinitely many occurrences of C, then it also contains infinitely many occurrences of $C\,t\,C'$.*

So, intuitively, if a run can execute a step infinitely often, it eventually will. It is readily seen that a fair run of a finite-state transition system eventually gets "trapped" in one of its bottom strongly connected components, and visits each of its states infinitely often. Hence, fair runs of a finite-state Markov chain have probability one. The following proposition was proved in [35] for a model slightly less general than replicated systems; the proof can be generalized without effort:

Proposition 1 ([35, Prop. 7])**.** *Let $\mathcal{P}$ be a replicated system, let C be a configuration of $\mathcal{P}$, and let φ be an LTL formula. It is the case that $\Pr[C, \varphi] = 1$ iff every fair run of $\mathcal{P}$ starting at C satisfies φ.*

We implicitly use this proposition from now on. In particular, we define:

Definition 2. *A configuration C* satisfies *φ with probability 1, or just* satisfies *φ, if every fair run starting at C satisfies φ, denoted by $C \models \varphi$. We let $[\![\varphi]\!]$ denote the set of configurations satisfying φ. A set $\mathcal{C}$ of configurations* satisfies *φ if $\mathcal{C} \subseteq [\![\varphi]\!]$, i.e., if $C \models \varphi$ for every $C \in \mathcal{C}$.*

Liveness Specifications for Replicated Systems. We focus on a specific class of temporal properties for which the qualitative model checking problem is decidable and which is large enough to formalize many important specifications. Using well-known automata-theoretic technology, this class can also be used to verify all properties describable in action-based LTL, see e.g. [35].

A *stable termination property* is given by a pair $\Pi = (\varphi_{\mathrm{pre}}, \Phi_{post})$, where $\Phi_{post} = \{\varphi^1_{\mathrm{post}}, \ldots, \varphi^k_{\mathrm{post}}\}$ and $\varphi_{\mathrm{pre}}, \varphi^1_{\mathrm{post}}, \ldots, \varphi^k_{\mathrm{post}}$ are Presburger formulas over Q describing sets of configurations. Whenever $k = 1$, we sometimes simply write $\Pi = (\varphi_{\mathrm{pre}}, \varphi_{\mathrm{post}})$. The pair Π induces the LTL property

$$\varphi_\Pi \stackrel{\mathrm{def}}{=} \Diamond \bigvee_{i=1}^{k} \Box \varphi^i_{\mathrm{post}}.$$

Abusing language, we say that a replicated system $\mathcal{P}$ *satisfies* Π if $[\![\varphi_{\mathrm{pre}}]\!] \subseteq [\![\varphi_\Pi]\!]$, that is, if every configuration C satisfying φ_{pre} satisfies φ_Π with probability 1. The *stable termination problem* is the qualitative model checking problem for $\mathcal{I} = [\![\varphi_{\mathrm{pre}}]\!]$ and $\varphi = \varphi_\Pi$ given by a stable termination property $\Pi = (\varphi_{\mathrm{pre}}, \Phi_{post})$.

Example 2. Let us reconsider the system from Example 1. We can formally specify that all agents will eventually agree on the majority output *Yes* or *No*. Let $\Pi^{\mathrm{Y}} = (\varphi^{\mathrm{Y}}_{\mathrm{pre}}, \varphi^{\mathrm{Y}}_{\mathrm{post}})$ and $\Pi^{\mathrm{N}} = (\varphi^{\mathrm{N}}_{\mathrm{pre}}, \varphi^{\mathrm{N}}_{\mathrm{post}})$ be defined by:

$$\varphi^{\mathrm{Y}}_{\mathrm{pre}} = (\mathrm{A_Y} > \mathrm{A_N} \wedge \mathrm{P_Y} + \mathrm{P_N} = 0), \qquad \varphi^{\mathrm{Y}}_{\mathrm{post}} = (\mathrm{A_N} + \mathrm{P_N} = 0),$$
$$\varphi^{\mathrm{N}}_{\mathrm{pre}} = (\mathrm{A_Y} \leq \mathrm{A_N} \wedge \mathrm{P_Y} + \mathrm{P_N} = 0), \qquad \varphi^{\mathrm{N}}_{\mathrm{post}} = (\mathrm{A_Y} + \mathrm{P_Y} = 0).$$

The system satisfies the property specified in Example 1 iff it satisfies Π^{Y} and Π^{N}. As an alternative (weaker) property, we could specify that the system always stabilizes to either output by $\Pi = (\varphi^{\mathrm{Y}}_{\mathrm{pre}} \vee \varphi^{\mathrm{N}}_{\mathrm{pre}}, \{\varphi^{\mathrm{Y}}_{\mathrm{post}}, \varphi^{\mathrm{N}}_{\mathrm{post}}\})$. ◁

3 Stage Graphs

In the rest of the paper, we fix a replicated system $\mathcal{P} = (Q, T)$ and a stable termination property $\Pi = (\varphi_{\text{pre}}, \Phi_{post})$, where $\Phi_{post} = \{\varphi^1_{\text{post}}, \ldots, \varphi^k_{\text{post}}\}$, and address the problem of checking whether $\mathcal{P}$ satisfies Π. We start with some basic definitions on sets of configurations.

Definition 3 (inductive sets, leads to, certificates)

- *A set of configurations $\mathcal{C}$ is* inductive *if $C \in \mathcal{C}$ and $C \to C'$ implies $C' \in \mathcal{C}$.*
- *Let $\mathcal{C}, \mathcal{C}'$ be sets of configurations. We say that $\mathcal{C}$* leads to *$\mathcal{C}'$, denoted $\mathcal{C} \rightsquigarrow \mathcal{C}'$, if for all $C \in \mathcal{C}$, every fair run from C eventually visits a configuration of $\mathcal{C}'$.*
- *A* certificate *for $\mathcal{C} \rightsquigarrow \mathcal{C}'$ is a function $f \colon \mathcal{C} \to \mathbb{N}$ satisfying that for every $C \in \mathcal{C} \setminus \mathcal{C}'$, there exists an execution $C \xrightarrow{*} C'$ such that $f(C) > f(C')$.*

Note that certificates only require the existence of some executions decreasing f, not for all of them to to decrease it. Despite this, we have:

Proposition 2. *For all inductive sets $\mathcal{C}, \mathcal{C}'$ of configurations, it is the case that: $\mathcal{C}$ leads to $\mathcal{C}'$ iff there exists a certificate for $\mathcal{C} \rightsquigarrow \mathcal{C}'$.*

The proof, which can be found in the full version [19], depends on two properties of replicated systems with stochastic scheduling. First, every configuration has only finitely many descendants. Second, for every fair run and for every finite execution $C \xrightarrow{w} C'$, if C appears infinitely often in the run, then the run contains infinitely many occurrences of $C \xrightarrow{w} C'$. We can now introduce stage graphs:

Definition 4 (stage graph). *A* stage graph *of $\mathcal{P}$ for the property Π is a directed acyclic graph whose nodes, called* stages*, are sets of configurations satisfying the following conditions:*

1. *every stage is an inductive set;*
2. *every configuration of $[\![\varphi_{\text{pre}}]\!]$ belongs to some stage;*
3. *if $\mathcal{C}$ is a non-terminal stage with successors $\mathcal{C}_1, \ldots, \mathcal{C}_n$, then there exists a certificate for $\mathcal{C} \rightsquigarrow (\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n)$;*
4. *if $\mathcal{C}$ is a terminal stage, then $\mathcal{C} \models \varphi^i_{\text{post}}$ for some i.*

The existence of a stage graph implies that $\mathcal{P}$ satisfies Π. Indeed, by conditions 2–3 and repeated application of Proposition 2, every run starting at a configuration of $[\![\varphi_{\text{pre}}]\!]$ eventually reaches a terminal stage, say $\mathcal{C}$, and, by condition 1, stays in $\mathcal{C}$ forever. Since, by condition 4, all configurations of $\mathcal{C}$ satisfy some φ^i_{post}, after its first visit to $\mathcal{C}$ every configuration satisfies φ^i_{post}.

Example 3. Figure 1 depicts stage graphs for the system of Example 1 and the properties defined in Example 2. The reader can easily show that every stage $\mathcal{C}$ is inductive by checking that for every $C \in \mathcal{C}$ and every transition $t \in \{t_1, \ldots, t_4\}$ enabled at C, the step $C \xrightarrow{t_i} C'$ satisfies $C' \in \mathcal{C}$. For example, if a configuration satisfies $\mathrm{A_Y} > \mathrm{A_N}$, so does any successor configuration. ◁

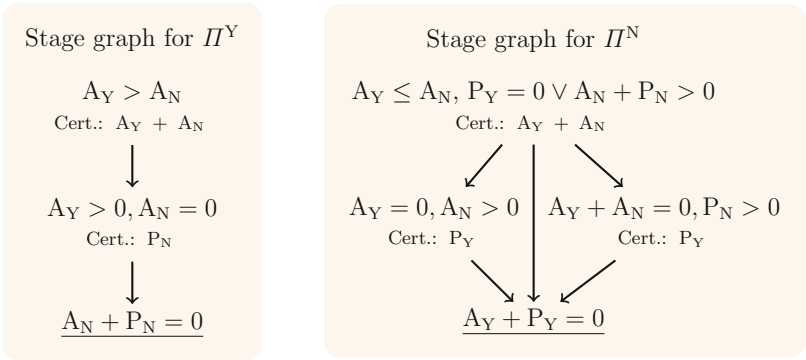


Fig. 1. Stage graphs for the system of Example 1.

The following proposition shows that stage graphs are a sound and complete technique for proving stable termination properties.

Proposition 3. *System $\mathcal{P}$ satisfies Π iff it has a stage graph for Π.*

Proposition 3 does not tell us anything about the decidability of the stable termination problem. To prove that the problem is decidable, we introduce Presburger stage graphs. Intuitively these are stage graphs whose stages and certificates can be expressed by formulas of Presburger arithmetic.

Definition 5 (Presburger stage graphs)

- *A stage $\mathcal{C}$ is* Presburger *if $\mathcal{C} = [\![\phi]\!]$ for some Presburger formula ϕ.*
- *A* bounded certificate *for $\mathcal{C} \rightsquigarrow \mathcal{C}'$ is a pair (f, k), where $f \colon \mathcal{C} \to \mathbb{N}$ and $k \in \mathbb{N}$, satisfying that for every $C \in \mathcal{C} \setminus \mathcal{C}'$, there exists an execution $C \xrightarrow{w} C'$ such that $f(C) > f(C')$ and $|w| \leq k$.*
- *A* Presburger certificate *is a bounded certificate (f, k) satisfying $f(C) = n \iff \varphi(C, n)$ for some Presburger formula $\varphi(\boldsymbol{x}, y)$.*
- *A* Presburger stage graph *is a stage graph whose stages and certificates are all Presburger.*

Using a powerful result from [36], we show that: (1) $\mathcal{P}$ satisfies Π iff it has a Presburger stage graph for Π (Theorem 2); (2) there exists a denumerable set of candidates for a Presburger stage graph for Π; and (3) there is an algorithm that decides whether a given candidate is a Presburger stage graph for Π (Theorem 3). Together, (1–3) show that the stable termination problem is semi-decidable. To obtain decidability, we observe that the complement of the stable termination problem is also semi-decidable. Indeed, it suffices to enumerate all initial configurations $C \models \varphi_{\mathrm{pre}}$, build for each such C the (finite) graph G_C of configurations reachable from C, and check if some bottom strongly connected component $\mathcal{B}$ of G_C satisfies $\mathcal{B} \not\models \varphi^i_{\mathrm{post}}$ for all i. This is the case iff some fair run starting at C visits and stays in $\mathcal{B}$, which in turn is the case iff $\mathcal{P}$ violates Π.

Theorem 2. *System $\mathcal{P}$ satisfies Π iff it has a Presburger stage graph for Π.*

We observe that testing whether a given graph is a Presburger stage graph reduces to Presburger arithmetic satisfiability, which is decidable [62] and whose complexity lies between 2-NEXP and 2-EXPSPACE [15]:

Theorem 3. *The problem of deciding whether an acyclic graph of Presburger sets and Presburger certificates is a Presburger stage graph, for a given stable termination property, is reducible in polynomial time to the satisfiability problem for Presburger arithmetic.*

4 Algorithmic Construction of Stage Graphs

At the current state of our knowledge, the decision procedure derived from Theorem 3 has little practical relevance. From a theoretical point of view, the TOWER-hardness result of [33] implies that the stage graph may have non-elementary size in the system size. In practice, systems have relatively small stage graphs, but, even so, the enumeration of all candidates immediately leads to a prohibitive combinatorial explosion.

For this reason, we present a procedure to automatically *construct* (not guess) a Presburger stage graph G for a given replicated system $\mathcal{P}$ and a stable termination property $\Pi = (\varphi_{\text{pre}}, \Phi_{post})$. The procedure may *fail*, but, as shown in the experimental section, it succeeds for many systems from the literature.

The procedure is designed to be implemented on top of a solver for the existential fragment of Presburger arithmetic. While every formula of Presburger arithmetic has an equivalent formula within the existential fragment [32,62], quantifier-elimination may lead to a doubly-exponential blow-up in the size of the formula. Thus, it is important to emphasize that our procedure *never requires to eliminate quantifiers*: If the pre- and postconditions of Π are supplied as quantifier-free formulas, then all constraints of the procedure remain in the existential fragment.

We give a high-level view of the procedure (see Algorithm 1), which uses several functions, described in detail in the rest of the paper. The procedure maintains a workset *WS* of Presburger stages, represented by existential Presburger formulas. Initially, the only stage is an inductive Presburger overapproximation $\mathit{PotReach}(\llbracket \varphi_{\text{pre}} \rrbracket)$ of the configurations reachable from $\llbracket \varphi_{\text{pre}} \rrbracket$ (*PotReach* is an abbreviation for "potentially reachable"). Notice that we must necessarily use an overapproximation, since $\mathit{post}^*(\llbracket \varphi_{\text{pre}} \rrbracket)$ is not always expressible in Presburger arithmetic[3]. We use a refinement of the overapproximation introduced in [22,37], equivalent to the overapproximation of [24].

In its main loop (lines 2–9), Algorithm 1 picks a Presburger stage $\mathcal{S}$ from the workset, and processes it. First, it calls $\text{Terminal}(\mathcal{S}, \Phi_{post})$ to check if $\mathcal{S}$ is terminal, i.e., whether $\mathcal{S} \models \varphi^i_{\text{post}}$ for some $\varphi^i_{\text{post}} \in \Phi_{post}$. This reduces to checking

[3] This follows easily from the fact that $\mathit{post}^*(\psi)$ is not always expressible in Presburger arithmetic for vector addition systems, even if ψ denotes a single configuration [43].

Algorithm 1: procedure for the construction of stage graphs.

Input: replicated system $\mathcal{P} = (Q, T)$, stable term. property $\Pi = (\varphi_{\text{pre}}, \Phi_{post})$
Result: a stage graph of $\mathcal{P}$ for Π

```
1  WS ← {PotReach([[φ_pre]])}
2  while WS ≠ ∅ do
3  |  remove S from WS
4  |  if ¬Terminal(S, Φ_post) then
5  |  |  U ← AsDead(S)
6  |  |  if U ≠ ∅ then
7  |  |  |  WS ← WS ∪ {IndOverapprox(S, U)}
8  |  |  else
9  |  |  |  WS ← WS ∪ Split(S)
```

the unsatisfiability of the existential Presburger formula $\phi \wedge \neg\varphi^i_{\text{post}}$, where ϕ is the formula characterizing $\mathcal{S}$. If $\mathcal{S}$ is not terminal, then the procedure attempts to construct successor stages in lines 5–9, with the help of three further functions: *AsDead*, *IndOverapprox*, and *Split*. In the rest of this section, we present the intuition behind lines 5–9, and the specification of the three functions. Sections 5, 6 and 7 present the implementations we use for these functions.

Lines 5–9 are inspired by the behavior of most replicated systems designed by humans, and are based on the notion of *dead* transitions, which can never occur again (to be formally defined below). Replicated systems are usually designed to run in *phases*. Initially, all transitions are alive, and the end of a phase is marked by the "death" of one or more transitions, i.e., by reaching a configuration at which these transitions are dead. The system keeps "killing transitions" until no transition that is still alive can lead to a configuration violating the postcondition. The procedure mimics this pattern. It constructs stage graphs in which if $\mathcal{S}'$ is a successor of $\mathcal{S}$, then the set of transitions dead at $\mathcal{S}'$ is a *proper superset* of the transitions dead at $\mathcal{S}$. For this, $AsDead(\mathcal{S})$ computes a set of transitions that are alive at some configuration of $\mathcal{S}$, but which will become dead in every fair run starting at $\mathcal{S}$ (line 5). Formally, $AsDead(\mathcal{S})$ returns a set $U \subseteq \overline{Dead(\mathcal{S})}$ such that $\mathcal{S} \models \Diamond\text{dead}(U)$, defined as follows.

Definition 6. *A transition of a replicated system $\mathcal{P}$ is* dead *at a configuration C if it is disabled at every configuration reachable from C (including C itself). A transition is* dead *at a stage $\mathcal{S}$ if it is dead at every configuration of $\mathcal{S}$. Given a stage $\mathcal{S}$ and a set U of transitions, we use the following notations:*

- *$Dead(\mathcal{S})$: the set of transitions dead at $\mathcal{S}$;*
- *$[\![dis(U)]\!]$: the set of configurations at which all transitions of U are disabled;*
- *$[\![dead(U)]\!]$: the set of configurations at which all transitions of U are dead.*

Observe that we can compute $Dead(\mathcal{S})$ by checking unsatisfiability of a sequence of existential Presburger formulas: as $\mathcal{S}$ is inductive, we have $Dead(\mathcal{S}) =$

$\{t \mid \mathcal{S} \models \text{dis}(t)\}$, and $\mathcal{S} \models \text{dis}(t)$ holds iff the existential Presburger formula $\exists C\colon \phi(C) \wedge C \geq {}^\bullet t$ is unsatisfiable, where ϕ is the formula characterizing $\mathcal{S}$.

The following proposition, whose proof appears in the full version [19], shows that determining whether a given transition will eventually become dead, while decidable, is PSPACE-hard. Therefore, Sect. 7 describes two implementations of this function, and a way to combine them, which exhibit a good trade-off between precision and computation time.

Proposition 4. *Given a replicated system $\mathcal{P}$, a stage $\mathcal{S}$ represented by an existential Presburger formula ϕ and a set of transitions U, determining whether $\mathcal{S} \models \Diamond \mathit{dead}(U)$ holds is decidable and PSPACE-hard.*

If the set U returned by $\mathit{AsDead}(\mathcal{S})$ is nonempty, then we know that every fair run starting at a configuration of $\mathcal{S}$ will eventually reach a configuration of $\mathcal{S} \cap [\![\text{dead}(U)]\!]$. So, this set, or any inductive overapproximation of it, can be a legal successor of $\mathcal{S}$ in the stage graph. Function $\mathit{IndOverapprox}(\mathcal{S}, U)$ returns such an inductive overapproximation (line 7). To be precise, we show in Sect. 5 that $[\![\text{dead}(U)]\!]$ is a Presburger set that can be computed exactly, albeit in doubly-exponential time in the worst case. The section also shows how to compute overapproximations more efficiently. If the set U returned by $\mathit{AsDead}(\mathcal{S})$ is empty, then we cannot yet construct any successor of $\mathcal{S}$. Indeed, recall that we want to construct stage graphs in which if $\mathcal{S}'$ is a successor of $\mathcal{S}$, then $\mathit{Dead}(\mathcal{S}')$ is a *proper superset* of $\mathit{Dead}(\mathcal{S})$. In this case, we proceed differently and try to split $\mathcal{S}$:

Definition 7. *A* split *of some stage $\mathcal{S}$ is a set $\{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$ of (not necessarily disjoint) stages such that the following holds:*

- $\mathit{Dead}(\mathcal{S}_i) \supset \mathit{Dead}(\mathcal{S})$ *for every* $1 \leq i \leq k$, *and*
- $\mathcal{S} = \bigcup_{i=1}^{k} \mathcal{S}_i$.

If there exists a split $\{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$ of $\mathcal{S}$, then we can let $\mathcal{S}_1, \ldots, \mathcal{S}_k$ be the successors of $\mathcal{S}$ in the stage graph. Observe that a stage may indeed have a split. We have $\mathit{Dead}(\mathcal{C}_1 \cup \mathcal{C}_2) = \mathit{Dead}(\mathcal{C}_1) \cap \mathit{Dead}(\mathcal{C}_2)$, and hence $\mathit{Dead}(\mathcal{C}_1 \cup \mathcal{C}_2)$ may be a proper subset of both $\mathit{Dead}(\mathcal{C}_1)$ and $\mathit{Dead}(\mathcal{C}_2)$:

Example 4. Consider the system with states $\{q_1, q_2\}$ and transitions $t_i\colon q_i \mapsto q_i$ for $i \in \{1, 2\}$. Let $\mathcal{S} = \{C \mid C(q_1) = 0 \vee C(q_2) = 0\}$, i.e., $\mathcal{S}$ is the (inductive) stage of configurations disabling either t_1 or t_2. The set $\{\mathcal{S}_1, \mathcal{S}_2\}$, where $\mathcal{S}_i = \{C \in \mathcal{S} \mid C(q_i) = 0\}$, is a split of $\mathcal{S}$ satisfying $\mathit{Dead}(\mathcal{S}_i) = \{t_i\} \supset \emptyset = \mathit{Dead}(\mathcal{S})$. ◁

The canonical split of $\mathcal{S}$, if it exists, is the set $\{\mathcal{S} \cap [\![\text{dead}(t)]\!] \mid t \notin \mathit{Dead}(\mathcal{S})\}$. As mentioned above, Sect. 5 shows that $[\![\text{dead}(U)]\!]$ can be computed exactly for every U, but the computation can be expensive. Hence, the canonical split can be computed exactly at potentially high cost. Our implementation uses an underapproximation of $[\![\text{dead}(t)]\!]$, described in Sect. 6.

5 Computing and Approximating $[\![\text{dead}(U)]\!]$

We show that, given a set U of transitions,

- we can effectively compute an existential Presburger formula describing the set $[\![\text{dead}(U)]\!]$, with high computational cost in the worst case, and
- we can effectively compute constraints that overapproximate or underapproximate $[\![\text{dead}(U)]\!]$, at a reduced computational cost.

Downward and Upward Closed Sets. We enrich $\mathbb{N}$ with the limit element ω in the usual way. In particular, $n < \omega$ holds for every $n \in \mathbb{N}$. An *ω-configuration* is a mapping $C^\omega \colon Q \to \mathbb{N} \cup \{\omega\}$. The *upward closure* and *downward closure* of a set $\mathcal{C}^\omega$ of ω-configurations are the sets of configurations $\uparrow \mathcal{C}^\omega$ and $\downarrow \mathcal{C}^\omega$, respectively defined as:

$$\uparrow \mathcal{C}^\omega \stackrel{\text{def}}{=} \{C \in \mathbb{N}^Q \mid C \geq C^\omega \text{ for some } C^\omega \in \mathcal{C}^\omega\},$$
$$\downarrow \mathcal{C}^\omega \stackrel{\text{def}}{=} \{C \in \mathbb{N}^Q \mid C \leq C^\omega \text{ for some } C^\omega \in \mathcal{C}^\omega\}.$$

A set $\mathcal{C}$ of configurations is *upward closed* if $\mathcal{C} = \uparrow \mathcal{C}$, and *downward closed* if $\mathcal{C} = \downarrow \mathcal{C}$. These facts are well-known from the theory of well-quasi orderings:

Lemma 1. *For every set $\mathcal{C}$ of configurations, the following holds:*

1. *$\mathcal{C}$ is upward closed iff $\overline{\mathcal{C}}$ is downward closed (and vice versa);*
2. *if $\mathcal{C}$ is upward closed, then there is a unique minimal finite set of configurations $\mathit{inf}(\mathcal{C})$, called its* basis*, such that $\mathcal{C} = \uparrow \mathit{inf}(\mathcal{C})$;*
3. *if $\mathcal{C}$ is downward closed, then there is a unique minimal finite set of ω-configurations $\mathit{sup}(\mathcal{C})$, called its* decomposition*, such that $\mathcal{C} = \downarrow \mathit{sup}(\mathcal{C})$.*

Computing $[\![\text{dead}(U)]\!]$ Exactly. It follows immediately from Definition 6 that both $[\![\text{dis}(U)]\!]$ and $[\![\text{dead}(U)]\!]$ are downward closed. Indeed, if all transitions of U are disabled at C, and $C' \leq C$, then they are also disabled at C', and clearly the same holds for transitions dead at C. Furthermore:

Proposition 5. *For every set U of transitions, the (downward) decomposition of both $\mathit{sup}([\![\mathit{dis}(U)]\!])$ and $\mathit{sup}([\![\mathit{dead}(U)]\!])$ is effectively computable.*

Proof. For every $t \in U$ and $q \in {}^\bullet t$, let $C^\omega_{t,q}$ be the ω-configuration such that $C^\omega_{t,q}(q) = {}^\bullet t(q) - 1$ and $C^\omega_{t,q}(p) = \omega$ for every $p \in Q \setminus \{q\}$. In other words, $C^\omega_{t,q}$ is the ω-configuration made only of ω's except for state q which falls short from ${}^\bullet t(q)$ by one. This ω-configurations captures all configurations disabled in t due to an insufficient amount of agents in state q. We have:

$$\sup([\![\text{dis}(U)]\!]) = \{C^\omega_{t,q} : t \in U, q \in {}^\bullet t\}.$$

The latter can be made minimal by removing superfluous ω-configurations.

For the case of $\sup([\![\text{dead}(U)]\!])$, we invoke [45, Prop. 2] which gives a proof for the more general setting of (possibly unbounded) Petri nets. Their procedure is based on the well-known backwards reachability algorithm (see, e.g., [2,39]). □

Since $\sup(\llbracket \text{dead}(U) \rrbracket)$ is finite, its computation allows to describe $\llbracket \text{dead}(U) \rrbracket$ by the following linear constraint[4]:

$$\bigvee_{C^\omega \in \sup(\llbracket \text{dead}(U) \rrbracket)} \bigwedge_{q \in Q} [C(q) \leq C^\omega(q)] \,.$$

However, the cardinality of $\sup(\llbracket \text{dead}(U) \rrbracket)$ can be exponential [45, Remark for Prop. 2] in the system size. For this reason, we are interested in constructing both under- and over-approximations.

Overapproximations of $\llbracket \text{dead}(U) \rrbracket$. For every $i \in \mathbb{N}$, define $\llbracket \text{dead}(U) \rrbracket^i$ as:

$$\llbracket \text{dead}(U) \rrbracket^0 \stackrel{\text{def}}{=} \llbracket \text{dis}(U) \rrbracket \quad \text{and} \quad \llbracket \text{dead}(U) \rrbracket^{i+1} \stackrel{\text{def}}{=} \overline{pre_T(\overline{\llbracket \text{dead}(U) \rrbracket^i})} \cap \llbracket \text{dis}(U) \rrbracket.$$

Loosely speaking, $\llbracket \text{dead}(U) \rrbracket^i$ is the set of configurations C such that every configuration reachable in at most i steps from C disables U. We immediately have:

$$\llbracket \text{dead}(U) \rrbracket = \bigcap_{i=0}^{\infty} \llbracket \text{dead}(U) \rrbracket^i.$$

Using Proposition 5 and the following proposition, we obtain that $\llbracket \text{dead}(U) \rrbracket^i$ is an effectively computable overapproximation of $\llbracket \text{dead}(U) \rrbracket$.

Proposition 6. *For every Presburger set $\mathcal{C}$ and every set of transitions U, the sets $pre_U(\mathcal{C})$ and $post_U(\mathcal{C})$ are effectively Presburger.*

Recall that function $\mathit{IndOverapprox}(\mathcal{S}, U)$ of Algorithm 1 must return an *inductive* overapproximation of $\llbracket \text{dead}(U) \rrbracket$. Since $\llbracket \text{dead}(U) \rrbracket^i$ might not be inductive in general, our implementation uses either the inductive overapproximations $\mathit{IndOverapprox}^i(\mathcal{S}, U) \stackrel{\text{def}}{=} \mathit{PotReach}(\mathcal{S} \cap \llbracket \text{dead}(U) \rrbracket^i)$, or the exact value $\mathit{IndOverapprox}^\infty(\mathcal{S}, U) \stackrel{\text{def}}{=} \mathcal{S} \cap \llbracket \text{dead}(U) \rrbracket$. The table of results in the experimental section describes for each benchmark which overapproximation was used.

Underapproximations of $\llbracket \text{dead}(U) \rrbracket$: Death Certificates. A *death certificate* for U in $\mathcal{P}$ is a finite set $\mathcal{C}^\omega$ of ω-configurations such that:

1. $\downarrow \mathcal{C}^\omega \models \text{dis}(U)$, i.e., every configuration of $\downarrow \mathcal{C}^\omega$ disables U, and
2. $\downarrow \mathcal{C}^\omega$ is inductive, i.e., $post_T(\downarrow \mathcal{C}^\omega) \subseteq \downarrow \mathcal{C}^\omega$.

If U is dead at a set $\mathcal{C}$ of configurations, then there is always a certificate that proves it, namely $\sup(\llbracket \text{dead}(U) \rrbracket)$. In particular, if $\mathcal{C}^\omega$ is a death certificate for U then $\downarrow \mathcal{C}^\omega \subseteq \llbracket \text{dead}(U) \rrbracket$, that is, $\downarrow \mathcal{C}^\omega$ is an underapproximation of $\llbracket \text{dead}(U) \rrbracket$

Using Proposition 6, it is straightforward to express in Presburger arithmetic that a finite set $\mathcal{C}^\omega$ of ω-configurations is a death certificate for U:

Proposition 7. *For every $k \geq 1$ there is an existential Presburger formula $\mathit{DeathCert}_k(U, \mathcal{C}^\omega)$ that holds iff $\mathcal{C}^\omega$ is a death certificate of size k for U.*

[4] Observe that if $C^\omega(q) = \omega$, then the term "$C(q) \leq \omega$" is equivalent to "**true**".

6 Splitting a Stage

Given a stage $\mathcal{S}$, we try to find a set $\mathcal{C}_1^\omega, \ldots, \mathcal{C}_\ell^\omega$ of death certificates for transitions $t_1, \ldots, t_\ell \in T \setminus \mathit{Dead}(\mathcal{S})$ such that $\mathcal{S} \subseteq \downarrow\mathcal{C}_1^\omega \cup \cdots \cup \downarrow\mathcal{C}_\ell^\omega$. This allows us to split $\mathcal{S}$ into $\mathcal{S}_1, \ldots, \mathcal{S}_\ell$, where $\mathcal{S}_i \stackrel{\text{def}}{=} \mathcal{S} \cap \downarrow\mathcal{C}_i^\omega$.

For any fixed size $k \geq 1$ and any fixed ℓ, we can find death certificates $\mathcal{C}_1^\omega, \ldots, \mathcal{C}_\ell^\omega$ of size at most k by solving a Presburger formula. However, the formula does not belong to the existential fragment, because the inclusion check $\mathcal{S} \subseteq \downarrow\mathcal{C}_1^\omega \cup \cdots \cup \downarrow\mathcal{C}_\ell^\omega$ requires universal quantification. For this reason, we proceed iteratively. For every $i \geq 0$, after having found $\mathcal{C}_1^\omega, \ldots, \mathcal{C}_i^\omega$ we search for a pair $(C_{i+1}, \mathcal{C}_{i+1}^\omega)$ such that

(i) $\mathcal{C}_{i+1}^\omega$ is a death certificate for some $t_{i+1} \in T \setminus \mathit{Dead}(\mathcal{S})$;
(ii) $C_{i+1} \in \mathcal{S} \cap \downarrow\mathcal{C}_{i+1}^\omega \setminus (\downarrow\mathcal{C}_1^\omega \cup \cdots \cup \downarrow\mathcal{C}_i^\omega)$.

An efficient implementation requires to guide the search for $(C_{i+1}, \mathcal{C}_{i+1}^\omega)$, because otherwise the search procedure might not even terminate, or might split $\mathcal{S}$ into too many parts, blowing up the size of the stage graph. Our search procedure employs the following heuristic, which works well in practice. We only consider the case $k = 1$, and search for a pair $(C_{i+1}, C_{i+1}^\omega)$ satisfying (i) and (ii) above, and additionally:

(iii) all components of C_{i+1}^ω are either ω or between 0 and $\max_{t \in T, q \in Q} {}^\bullet t(q) - 1$;
(iv) for every ω-configuration C^ω, if (C_{i+1}, C^ω) satisfies (i)–(iii), then $C_{i+1}^\omega \leq C^\omega$;
(v) for every pair (C, C^ω), if (C, C^ω) satisfies (i)–(iv), then $C^\omega \leq C_{i+1}^\omega$.

Condition (iii) guarantees termination. Intuitively, condition (iv) leads to certificates valid for sets $U \subseteq T \setminus \mathit{Dead}(\mathcal{S})$ as large as possible. So it allows us to avoid splits that, loosely speaking, do not make as much progress as they could. Condition (v) allows us to avoid splits with many elements because each element of the split has a small intersection with $\mathcal{S}$.

An example illustrating these conditions is given in the full version [19].

7 Computing Eventually Dead Transitions

Recall that the function $\mathit{AsDead}(\mathcal{S})$ takes an inductive Presburger set $\mathcal{S}$ as input, and returns a (possibly empty) set $U \subseteq \overline{\mathit{Dead}(\mathcal{S})}$ of transitions such that $\mathcal{S} \models \Diamond\mathrm{dead}(U)$. This guarantees $\mathcal{S} \rightsquigarrow [\![\mathrm{dead}(U)]\!]$ and, since $\mathcal{S}$ is inductive, also $\mathcal{S} \rightsquigarrow \mathcal{S} \cap [\![\mathrm{dead}(U)]\!]$.

By Proposition 4, deciding if there exists a non-empty set U of transitions such that $\mathcal{S} \models \Diamond\mathrm{dead}(U)$ holds is PSPACE-hard, which makes a polynomial reduction to satisfiability of existential Presburger formulas unlikely. So we design incomplete implementations of $\mathit{AsDead}(\mathcal{S})$ with lower complexity. Combining these implementations, the lack of completeness essentially vanishes in practice.

The implementations are inspired by Proposition 2, which shows that $\mathcal{S} \rightsquigarrow [\![\mathrm{dead}(U)]\!]$ holds iff there exists a certificate f such that:

$$\forall C \in \mathcal{S} \setminus [\![\mathrm{dead}(U)]\!] : \exists C \xrightarrow{*} C' \colon f(C) > f(C'). \qquad \text{(Cert)}$$

To find such certificates efficiently, we only search for *linear* functions $f(C) = \sum_{q \in Q} \boldsymbol{a}(q) \cdot C(q)$ with coefficients $\boldsymbol{a}(q) \in \mathbb{N}$ for each $q \in Q$.

7.1 First Implementation: Linear Ranking Functions

Our first procedure computes the existence of a linear *ranking function*.

Definition 8. *A function $r \colon \mathcal{S} \to \mathbb{N}$ is a ranking function for $\mathcal{S}$ and U if for every $C \in \mathcal{S}$ and every step $C \xrightarrow{t} C'$ the following holds:*

1. *if $t \in U$, then $r(C) > r(C')$; and*
2. *if $t \notin U$, then $r(C) \geq r(C')$.*

Proposition 8. *If $r \colon \mathcal{S} \to \mathbb{N}$ is a ranking function for $\mathcal{S}$ and U, then there exists $k \in \mathbb{N}$ such that (r, k) is a bounded certificate for $\mathcal{S} \rightsquigarrow [\![dead(U)]\!]$.*

Proof. Let M be the minimal finite basis of the upward closed set $\overline{[\![\mathrm{dead}(U)]\!]}$. For every configuration $D \in M$, let σ_D be a shortest sequence that enables some transition of $t_D \in U$ from D, i.e., such that $D \xrightarrow{\sigma_D} D' \xrightarrow{t_D} D''$ for some D', D''. Let $k \stackrel{\text{def}}{=} \max\{|\sigma_D t_D| : D \in M\}$.

Let $C \in \mathcal{S} \setminus [\![\mathrm{dead}(U)]\!]$. Since $C \in \overline{[\![\mathrm{dead}(U)]\!]}$, we have $C \geq D$ for some $D \in M$. By monotonicity, we have $C \xrightarrow{\sigma_D} C' \xrightarrow{t_D} C''$ for some configurations C' and C''. By Definition 8, we have $r(C) \geq r(C') > r(C'')$, and so condition (Cert) holds. As $|\sigma_D t_D| \leq k$, we have that (r, k) is a bounded certificate. □

It follows immediately from Definition 8 that if r_1 and r_2 are ranking functions for sets U_1 and U_2 respectively, then r defined as $r(C) \stackrel{\text{def}}{=} r_1(C) + r_2(C)$ is a ranking function for $U_1 \cup U_2$. Therefore, there exists a unique maximal set of transitions U such that $\mathcal{S} \rightsquigarrow [\![\mathrm{dead}(U)]\!]$ can be proved by means of a ranking function. Further, U can be computed by collecting all transitions $t \in \overline{Dead(\mathcal{S})}$ such that there exists a ranking function r_t for $\{t\}$. The existence of a *linear* ranking function r_t can be decided in polynomial time via linear programming, as follows. Recall that for every step $C \xrightarrow{u} C'$, we have $C' = C + \Delta(u)$. So, by linearity, we have $r_t(C) \geq r_t(C') \iff r_t(C' - C) \leq 0 \iff r_t(\Delta(u)) \leq 0$. Thus, the constraints of Definition 8 can be specified as:

$$\boldsymbol{a} \cdot \Delta(t) < 0 \quad \wedge \bigwedge_{u \in \overline{Dead(\mathcal{S})}} \boldsymbol{a} \cdot \Delta(u) \leq 0,$$

where $\boldsymbol{a} \colon Q \to \mathbb{Q}_{\geq 0}$ gives the coefficients of r_t, that is, $r_t(C) = \boldsymbol{a} \cdot C$, and $\boldsymbol{a} \cdot \boldsymbol{x} \stackrel{\text{def}}{=} \sum_{q \in Q} \boldsymbol{a}(q) \cdot \boldsymbol{x}(q)$ for $\boldsymbol{x} \in \mathbb{N}^Q$. Observe that a solution may yield a function whose codomain differs from $\mathbb{N}$. However, this is not an issue since we can scale it with the least common denominator of each $\boldsymbol{a}(q)$.

7.2 Second Implementation: Layers

Transitions layers were introduced in [22] as a technique to find transitions that will eventually become dead. Intuitively, a set U of transitions is a layer if (1) no run can contain only transitions of U, and (2) U becomes dead once disabled; the first condition guarantees that U eventually becomes disabled, and the second that it eventually becomes dead. We formalize layers in terms of *layer functions*.

Definition 9. *A function* $\ell\colon \mathcal{S} \to \mathbb{N}$ *is a* layer function *for* $\mathcal{S}$ *and* U *if:*

C1. $\ell(C) > \ell(C')$ *for every* $C \in \mathcal{S}$ *and every step* $C \xrightarrow{t} C'$ *with* $t \in U$*; and*
C2. $[\![dis(U)]\!] = [\![dead(U)]\!]$.

Proposition 9. *If* $\ell\colon \mathcal{S} \to \mathbb{N}$ *is a layer function for* $\mathcal{S}$ *and* U*, then* $(\ell, 1)$ *is a bounded certificate for* $\mathcal{S} \rightsquigarrow [\![dead(U)]\!]$.

Proof. Let $C \in \mathcal{S} \setminus [\![\mathrm{dead}(U)]\!]$. By condition **C2**, we have $C \notin [\![\mathrm{dis}(U)]\!]$. So there exists a step $C \xrightarrow{u} C'$ where $u \in U$. By condition **C1**, we have $\ell(C) > \ell(C')$, so condition (Cert) holds and $(\ell, 1)$ is a bounded certificate.

Let $\mathcal{S}$ be a stage. For every set of transitions $U \subseteq \overline{Dead(\mathcal{S})}$ we can construct a Presburger formula $\mathit{lin\text{-}layer}(U, \boldsymbol{a})$ that holds iff there there exists a *linear* layer function for U, i.e., a layer function of the form $\ell(C) = \boldsymbol{a} \cdot C$ for a vector of coefficients $\boldsymbol{a}\colon Q \to \mathbb{Q}_{\geq 0}$. Condition **C1**, for a linear function $\ell(C)$, is expressed by the existential Presburger formula

$$\mathit{lin\text{-}layer\text{-}fun}(U, \boldsymbol{a}) \stackrel{\text{def}}{=} \bigwedge_{u \in U} \boldsymbol{a} \cdot \Delta(u) < 0.$$

Condition **C2** is expressible in Presburger arithmetic because of Proposition 5. However, instead of computing $[\![\mathrm{dead}(U)]\!]$ explicitly, there is a more efficient way to express this constraint. Intuitively, $[\![\mathrm{dis}(U)]\!] = [\![\mathrm{dead}(U)]\!]$ is the case if enabling a transition $u \in U$ requires to have previously enabled some transition $u' \in U$. This observation leads to:

Proposition 10. *A set* U *of transitions satisfies* $[\![dis(U)]\!] = [\![dead(U)]\!]$ *iff it satisfies the existential Presburger formula*

$$\mathit{dis\text{-}eq\text{-}dead}(U) \stackrel{def}{=} \bigwedge_{t \in T} \bigwedge_{u \in U} \bigvee_{u' \in U} {}^\bullet t + ({}^\bullet u \ominus t^\bullet) \geq {}^\bullet u'$$

where $\boldsymbol{x} \ominus \boldsymbol{y} \in \mathbb{N}^Q$ *is defined by* $(\boldsymbol{x} \ominus \boldsymbol{y})(q) \stackrel{def}{=} \max(\boldsymbol{x}(q) - \boldsymbol{y}(q), 0)$ *for* $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{N}^Q$.

This allows us to give the constraint $\mathit{lin\text{-}layer}(U, \boldsymbol{a})$, which is of polynomial size:

$$\mathit{lin\text{-}layer}(U, \boldsymbol{a}) \stackrel{\text{def}}{=} \mathit{lin\text{-}layer\text{-}fun}(U, \boldsymbol{a}) \wedge \mathit{dis\text{-}eq\text{-}dead}(U).$$

7.3 Comparing Ranking and Layer Functions

The ranking and layer functions of Sects. 7.1 and 7.2 are incomparable in power, that is, there are sets of transitions for which a ranking function but no layer function exists, and vice versa. This is shown by the following two systems:

$$\mathcal{P}_1 = (\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}, \{t_1 \colon \mathrm{A}\,\mathrm{B} \mapsto \mathrm{C}\,\mathrm{C},\ t_2 \colon \mathrm{A} \mapsto \mathrm{B},\ t_3 \colon \mathrm{B} \mapsto \mathrm{A}\}),$$
$$\mathcal{P}_2 = (\{\mathrm{A}, \mathrm{B}\}, \quad \{t_4 \colon \mathrm{A}\,\mathrm{B} \mapsto \mathrm{A}\,\mathrm{A},\ t_5 \colon \mathrm{A} \mapsto \mathrm{B}\}).$$

Consider the system $\mathcal{P}_1$, and let $\mathcal{S} = \mathbb{N}^Q$, i.e., $\mathcal{S}$ contains all configurations. Transitions t_2 and t_3 never become dead at $\wr \mathrm{A} \wr$ and can thus never be included in any U. Transition t_1 eventually becomes dead, as shown by the linear ranking function $r(C) = C(\mathrm{A}) + C(\mathrm{B})$ for $U = \{t_1\}$. But for this U, the condition **C2** for layer functions is not satisfied, as $[\![\mathrm{dis}(U)]\!] \ni \wr \mathrm{A}, \mathrm{A} \wr \xrightarrow{t_2} \wr \mathrm{A}, \mathrm{B} \wr \notin [\![\mathrm{dis}(U)]\!]$, so $[\![\mathrm{dis}(U)]\!] \neq [\![\mathrm{dead}(U)]\!]$. Therefore no layer function exists for this U.

Consider now the system $\mathcal{P}_2$, again with $\mathcal{S} = \mathbb{N}^Q$, and let $U = \{t_5\}$. Once t_5 is disabled, there is no agent in A, so both t_4 and t_5 are dead. So $[\![\mathrm{dis}(U)]\!] = [\![\mathrm{dead}(U)]\!]$. The linear layer function $\ell(C) = C(\mathrm{A})$ satisfies *lin-layer-fun*$(U, \boldsymbol{a})$, showing that U eventually becomes dead. As $C \xrightarrow{t_4 t_5} C$ for $C = \wr \mathrm{A}, \mathrm{B} \wr$, there is no ranking function r for this U, which would need to satisfy $r(C) < r(C)$.

For our implementation of *AsDead*$(\mathcal{S})$, we therefore combine both approaches. We first compute (in polynomial time) the unique maximal set U for which there is a linear ranking function. If this U is non-empty, we return it, and otherwise compute a set U of maximal size for which there is a linear layer function.

8 Experimental Results

We implemented the procedure of Sect. 4 on top of the SMT solver *Z3* [57], and use the Owl [48] and HOA [12] libraries for translating LTL formulas. The resulting tool automatically constructs stage graphs that verify stable termination properties for replicated systems. We evaluated it on two sets of benchmarks, described below. The first set contains population protocols, and the second leader election and mutual exclusion algorithms. All tests where performed on a machine with an Intel Xeon CPU E5-2630 v4 @ 2.20 GHz and 8GB of RAM. The results are depicted in Fig. 2 and can be reproduced by the certified artifact [18]. For parametric families of replicated systems, we always report the largest instance that we were able to verify with a timeout of one hour. For *IndOverapprox*, from the approaches in Sect. 5, we use *IndOverapprox*^{0} in the examples marked with * and *IndOverapprox*$^\infty$ otherwise. Almost all constructed stage graphs are a chain with at most 3 stages. The only exceptions are the stage graphs for the approximate majority protocols that contained a binary split and 5 stages. The size of the Presburger formulas increases with increasing size of the replicated system. In the worst case, this growth can be exponential. However, the growth is linear in all examples marked with *.

Population protocols (correctness) **Parameters**	$\|Q\|$	$\|T\|$	**Time**
Broadcast [31,22] *			
	2	1	$< 1s$
Majority (Example 1)[22] *			
	4	4	$< 1s$
Majority [23, Ex. 3] *			
	5	6	$< 1s$
Majority [5] ("fast & exact")			
m=13, d=1	16	136	$4s$
m=21, d=1 (TO: 23,1)	24	300	$466s$
m=21, d=20 (TO: 23,22)	62	1953	$3301s$
Flock-of-birds [28,22] *: $x \geq c$			
$c = 20$	21	210	$5s$
$c = 40$	41	820	$45s$
$c = 60$	61	1830	$341s$
$c = 80$ (TO: $c = 90$)	81	3240	$1217s$
Flock-of-birds [20, Sect. 3]: $x \geq c$			
$c = 60$	8	18	$15s$
$c = 90$	9	21	$271s$
$c = 120$ (TO: $c = 127$)	9	21	$2551s$
Flock-of-birds [31,22, *threshold-n*] *: $x \geq c$			
$c = 10$	11	19	$< 1s$
$c = 15$	16	29	$1s$
$c = 20$ (TO: $c = 25$)	21	39	$18s$
Threshold [8][22, $v_{\max}$=$c+1$] *: $\boldsymbol{a} \cdot \boldsymbol{x} \geq c$			
$c = 2$	28	288	$7s$
$c = 4$	44	716	$26s$
$c = 6$	60	1336	$107s$
$c = 8$ (TO: $c = 10$)	76	2148	$1089s$
Threshold [20] ("succinct"): $\boldsymbol{a} \cdot \boldsymbol{x} \geq c$			
$c = 7$	13	37	$2s$
$c = 31$	17	55	$11s$
$c = 127$	21	73	$158s$
$c = 511$ (TO: $c = 1023$)	25	91	$2659s$
Remainder [22] *: $\boldsymbol{a} \cdot \boldsymbol{x} \equiv_m c$			
$m = 5$	7	20	$< 1s$
$m = 15$	17	135	$34s$
$m = 20$ (TO: $m = 25$)	22	230	$1646s$

Population protocols (stable cons.) **Parameters**	$\|Q\|$	$\|T\|$	**Time**
Approx. majority [27] (Cell cycle sw.) *			
	3	4	$< 1s$
Approx. majority [51] (Coin game) *			
$k = 3$	2	4	$< 1s$
Approx. majority [56] (Moran proc.) *			
	2	2	$< 1s$

Leader election/Mutex algorithms Processes	$\|Q\|$	$\|T\|$	**Time**
Leader election [44] (Israeli-Jalfon)			
20	40	80	$7s$
60	120	240	$1493s$
70 (TO: 80)	140	280	$3295s$
Leader election [42] (Herman)			
21	42	42	$9s$
51	102	102	$300s$
81 (TO: 91)	162	162	$2800s$
Mutex [40] (Array)			
2	15	95	$2s$
5	33	239	$5s$
10 (TO: 11)	63	479	$938s$
Mutex [59] (Burns)			
2	11	75	$1s$
4	19	199	$119s$
5 (TO: 6)	23	279	$2232s$
Mutex [3] (Dijkstra)			
2	19	196	$66s$
3 (TO: 4)	27	488	$3468s$
Mutex [50] (Lehmann Rabin)			
2	19	135	$3s$
5	43	339	$115s$
9 (TO: 10)	75	611	$2470s$
Mutex [61] (Peterson)			
2	13	86	$2s$
Mutex [64] (Szymanski)			
2	17	211	$10s$
3 (TO: 4)	24	895	$667s$

Fig. 2. Columns $|Q|$, $|T|$, and **Time** give the number of states and non-silent transitions, and the time for verification. Population protocols are verified for an infinite set of configurations. For parametric families, the smallest instance that could not be verified within one hour is shown in brackets, e.g. (TO: $c = 90$). Leader election and mutex algorithms are verified for one configuration. The number of processes leading to a timeout is given in brackets, e.g. (TO: 10).

Population Protocols. Population protocols [8,9] are replicated systems that compute Presburger predicates following the computation-as-consensus paradigm [10]. Depending on whether the initial configuration of agents satisfies the predicate or not, the agents of a correct protocol eventually agree on the output "yes" or "no", almost surely. Example 1 can be interpreted as a population protocol for the majority predicate $\mathrm{A_Y} > \mathrm{A_N}$, and the two stable termination properties that verify its correctness are described in Example 2. To show that a population protocol correctly computes a given predicate, we thus construct two Presburger stage graphs for the two corresponding stable termination properties. In all these examples, correctness is proved for an infinite set of initial configurations.

Our set of benchmarks contains a broadcast protocol [31], three majority protocols (Example 1, [23, Ex. 3], [5]), and multiple instances of parameterized families of protocols, where each protocol computes a different instance of a parameterized family of predicates[5]. These include various *flock-of-birds* protocol families ([28], [20, Sect. 3], [31, *threshold-n*]) for the family of predicates $x \geq c$ for some constant $c \geq 0$; two families for threshold predicates of the form $\boldsymbol{a} \cdot \boldsymbol{x} \geq c$ [8,20]; and one family for remainder protocols of the form $\boldsymbol{a} \cdot \boldsymbol{x} \equiv_m c$ [22]. Further, we check approximate majority protocols ([27,56], [51, *coin game*]). As these protocols only compute the predicate with large probability but not almost surely, we only verify that they always converge to a stable consensus.

Comparison with [22]. The approach of [22] can only be applied to so-called *strongly-silent* protocols. However, this class does not contain many fast and succinct protocols recently developed for different tasks [4,17,20].

We are able to verify all six protocols reported in [22]. Further, we are also able to verify the fast Majority [5] protocol as well as the succinct protocols Flock-of-birds [20, Sect. 3] and Threshold [20]. All three protocols are not strongly-silent. Although our approach is more general and complete, the time to verify many strongly-silent protocol does not differ significantly between the two approaches. Exceptions are the Flock-of-birds [28] protocols where we are faster ([22] reaches the timeout at $c = 55$) as well as the Remainder and the Flock-of-birds-threshold-n protocols where we are substantially slower ([22] reaches the timeout at $m = 80$ and $c = 350$, respectively). Loosely speaking, the approach of [22] can be faster because they compute inductive overapproximations using an iterative procedure instead of *PotReach*. In some instances already a very weak overapproximation, much less precise than *PotReach*, suffices to verify the result. Our procedure can be adapted to accommodate this (it essentially amounts to first running the procedure of [22], and if it is inconclusive then run ours).

Other Distributed Algorithms. We have also used our approach to verify arbitrary LTL liveness properties of non-parameterized systems with arbitrary communication structure. For this we apply standard automata-theoretic techniques and

[5] Notice that for each protocol we check correctness for all inputs; we cannot yet automatically verify that infinitely many protocols are correct, each of them for all possible inputs.

construct a product of the system and a *limit-deterministic Büchi automaton* for the negation of the property. Checking that no fair runs of the product are accepted by the automaton reduces to checking a stable termination property.

Since we only check correctness of one single finite-state system, we can also apply a probabilistic model checker based on state-space exploration. However, our technique delivers a stage graph, which plays two roles. First, it gives an explanation of why the property holds in terms of invariants and ranking functions, and second, it is a certificate of correctness that can be efficiently checked by independent means.

We verify liveness properties for several leader election and mutex algorithms from the literature [3,40,42,44,50,59,61,64] under the assumption of a probabilistic scheduler. For the leader election algorithms, we check that a leader is eventually chosen; for the mutex algorithms, we check that the first process enters its critical section infinitely often.

Comparison with PRISM [49]. We compared execution times for verification by our technique and by PRISM on the same models. While PRISM only needs a few seconds to verify instances of the mutex algorithms [3,40,50,59,61,64] where we reach the time limit, it reaches the memory limit for the two leader election algorithms [42,44] already for 70 and 71 processes, which we can still verify.

9 Conclusion and Further Work

We have presented stage graphs, a sound and complete technique for the verification of stable termination properties of replicated systems, an important class of parameterized systems. Using deep results of the theory of Petri nets, we have shown that Presburger stage graphs, a class of stage graphs whose correctness can be reduced to the satisfiability problem of Presburger arithmetic, are also sound and complete. This provides a decision procedure for the verification of termination properties, which is of theoretical nature since it involves a blind enumeration of candidates for Presburger stage graphs. For this reason, we have presented a technique for the algorithmic construction of Presburger stage graphs, designed to exploit the strengths of SMT-solvers for existential Presburger formulas, i.e., integer linear constraints. Loosely speaking, the technique searches for *linear* functions certifying the progress between stages, even though only the much larger class of Presburger functions guarantees completeness.

We have conducted extensive experiments on a large set of benchmarks. In particular, our approach is able to prove correctness of nearly all the standard protocols described in the literature, including several protocols that could not be proved by the technique of [22], which only worked for so-called strongly-silent protocols. We have also successfully applied the technique to some self-stabilization algorithms, leader election and mutual exclusion algorithms.

Our technique is based on the mechanized search for invariants and ranking functions. It avoids the use of state-space exploration as much as possible. For this reason, it also makes sense as a technique for the verification of liveness properties of non-parameterized systems with a finite but very large state space.

References

1. Abdulla, P.A.: Regular model checking. Int. J. Softw. Tools Technol. Transf. **14**(2), 109–118 (2012). https://doi.org/10.1007/s10009-011-0216-8
2. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, LICS 1996, New Brunswick, New Jersey, USA, 27–30 July 1996, pp. 313–321. IEEE Computer Society (1996). https://doi.org/10.1109/LICS.1996.561359
3. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_56
4. Alistarh, D., Gelashvili, R.: Recent algorithmic advances in population protocols. SIGACT News **49**(3), 63–73 (2018). https://doi.org/10.1145/3289137.3289150
5. Alistarh, D., Gelashvili, R., Vojnovic, M.: Fast and exact majority in population protocols. In: Georgiou, C., Spirakis, P.G. (eds.) Proceedings of the 34th ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, 21–23 July 2015, pp. 47–56. ACM (2015). https://doi.org/10.1145/2767386.2767429
6. Aminof, B., Rubin, S., Zuleger, F., Spegni, F.: Liveness of parameterized timed networks. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015, Part II. LNCS, vol. 9135, pp. 375–387. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_30
7. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
8. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: Chaudhuri, S., Kutten, S. (eds.) Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, 25–28 July 2004, pp. 290–299. ACM (2004). https://doi.org/10.1145/1011767.1011810
9. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. Distrib. Comput. **18**(4), 235–253 (2006). https://doi.org/10.1007/s00446-005-0138-3
10. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distrib. Comput. **20**(4), 279–304 (2007). https://doi.org/10.1007/s00446-007-0040-2
11. Athanasiou, K., Liu, P., Wahl, T.: Unbounded-thread program verification using thread-state equations. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 516–531. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_35
12. Babiak, T., et al.: The Hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 479–486. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_31
13. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
14. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 64–78. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_9

15. Berman, L.: The complexitiy of logical theories. Theoret. Comput. Sci. **11**, 71–77 (1980). https://doi.org/10.1016/0304-3975(80)90037-7
16. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers (2015). https://doi.org/10.2200/S00658ED1V01Y201508DCT013
17. Blondin, M., Esparza, J., Genest, B., Helfrich, M., Jaax, S.: Succinct population protocols for presburger arithmetic. In: Proceedings of 37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, 10–13 March 2020, Montpellier, France. LIPIcs, vol. 154, pp. 40:1–40:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.STACS.2020.40
18. Blondin, M., Esparza, J., Helfrich, M., Kučera, A., Meyer, P.J.: Artifact evaluation VM and instructions to generate experimental results for the CAV20 paper: checking Qualitative Liveness Properties of Replicated Systems with Stochastic Scheduling. figshare:12295982 (2020). https://doi.org/10.6084/m9.figshare.12295982.v2
19. Blondin, M., Esparza, J., Helfrich, M., Kučera, A., Meyer, P.J.: Checking qualitative liveness properties of replicated systems with stochastic scheduling. arXiv:2005.03555 [cs.LO] (2020). https://arxiv.org/abs/2005.03555
20. Blondin, M., Esparza, J., Jaax, S.: Large flocks of small birds: on the minimal size of population protocols. In: Proceedings of 35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, 28 February - 3 March 2018, Caen, France. LIPIcs, vol. 96, pp. 16:1–16:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.STACS.2018.16
21. Blondin, M., Esparza, J., Jaax, S.: Peregrine: a tool for the analysis of population protocols. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part I. LNCS, vol. 10981, pp. 604–611. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_34
22. Blondin, M., Esparza, J., Jaax, S., Meyer, P.J.: Towards efficient verification of population protocols. In: Schiller, E.M., Schwarzmann, A.A. (eds.) Proceedings of 36th ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, 25–27 July 2017, pp. 423–430. ACM (2017). https://doi.org/10.1145/3087801.3087816
23. Blondin, M., Esparza, J., Kučera, A.: Automatic analysis of expected termination time for population protocols. In: Schewe, S., Zhang, L. (eds.) Proceedings of 29th International Conference on Concurrency Theory, CONCUR 2018, 4–7 September 2018, Beijing, China. LIPIcs, vol. 118, pp. 33:1–33:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CONCUR.2018.33
24. Blondin, M., Finkel, A., Haase, C., Haddad, S.: The logical view on continuous petri nets. ACM Trans. Comput. Log. (TOCL) **18**(3), 24:1–24:28 (2017). https://doi.org/10.1145/3105908
25. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_31
26. Browne, M.C., Clarke, E.M., Grumberg, O.: Reasoning about networks with many identical finite state processes. Inf. Comput. **81**(1), 13–31 (1989). https://doi.org/10.1016/0890-5401(89)90026-6
27. Cardelli, L., Csikász-Nagy, A.: The cell cycle switch computes approximate majority. Sci. Rep. **2**(1), 656 (2012). https://doi.org/10.1038/srep00656
pagebreak

28. Chatzigiannakis, I., Michail, O., Spirakis, P.G.: Algorithmic verification of population protocols. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 221–235. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16023-3_19
29. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: Stewart, D., Weissenbacher, G. (eds.) Proceedings of 17th International Conference on Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 76–83. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102244
30. Clarke, E., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_18
31. Clément, J., Delporte-Gallet, C., Fauconnier, H., Sighireanu, M.: Guidelines for the verification of population protocols. In: Proceedings of 31st International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, 20–24 June 2011, pp. 215–224. IEEE Computer Society (2011). https://doi.org/10.1109/ICDCS.2011.36
32. Cooper, D.C.: Theorem proving in arithmetic without multiplication. Mach. Intell. **7**, 91–99 (1972)
33. Czerwinski, W., Lasota, S., Lazic, R., Leroux, J., Mazowiecki, F.: The reachability problem for petri nets is not elementary. In: Charikar, M., Cohen, E. (eds.) Proceedings of 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, 23–26 June 2019, pp. 24–33. ACM (2019). https://doi.org/10.1145/3313276.3316369
34. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. **14**(4), 527–550 (2003). https://doi.org/10.1142/S0129054103001881
35. Esparza, J., Ganty, P., Leroux, J., Majumdar, R.: Model checking population protocols. In: Lal, A., Akshay, S., Saurabh, S., Sen, S. (eds.) Proceedings of 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, Chennai, India, 13–15 December 2016. LIPIcs, vol. 65, pp. 27:1–27:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.FSTTCS.2016.27
36. Esparza, J., Ganty, P., Leroux, J., Majumdar, R.: Verification of population protocols. Acta Inf. **54**(2), 191–215 (2017). https://doi.org/10.1007/s00236-016-0272-3
37. Esparza, J., Ledesma-Garza, R., Majumdar, R., Meyer, P., Niksic, F.: An SMT-based approach to coverability analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 603–619. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_40
38. Esparza, J., Meyer, P.J.: An SMT-based approach to fair termination analysis. In: Kaivola, R., Wahl, T. (eds.) Proceedings of 15th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, 27–30 September 2015, pp. 49–56. IEEE (2015)
39. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere!. Theoret. Comput. Sci. **256**(1–2), 63–92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X

40. Fribourg, L., Olsén, H.: Reachability sets of parameterized rings as regular languages. In: Moller, F. (ed.) Proceedings of 2nd International Workshop on Verification of Infinite State Systems, Infinity 1997, Bologna, Italy, 11–12 July 1997. Electronic Notes in Theoretical Computer Science, vol. 9, p. 40. Elsevier (1997). https://doi.org/10.1016/S1571-0661(05)80427-X
41. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992). https://doi.org/10.1145/146637.146681
42. Herman, T.: Probabilistic self-stabilization. Inf. Process. Lett. **35**(2), 63–67 (1990). https://doi.org/10.1016/0020-0190(90)90107-9
43. Hopcroft, J.E., Pansiot, J.: On the reachability problem for 5-dimensional vector addition systems. Theoret. Comput. Sci. **8**, 135–159 (1979). https://doi.org/10.1016/0304-3975(79)90041-0
44. Israeli, A., Jalfon, M.: Token management schemes and random walks yield self-stabilizing mutual exclusion. In: Dwork, C. (ed.) Proceedings of 9th Annual ACM Symposium on Principles of Distributed Computing, PODC 1990, Quebec City, Quebec, Canada, 22–24 August 1990, pp. 119–131. ACM (1990). https://doi.org/10.1145/93385.93409
45. Jancar, P., Purser, D.: Structural liveness of petri nets is expspace-hard and decidable. Acta Inf. **56**(6), 537–552 (2019). https://doi.org/10.1007/s00236-019-00338-6
46. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_55
47. Kaiser, A., Kroening, D., Wahl, T.: A widening approach to multithreaded program verification. ACM Trans. Program. Lang. Syst. **36**(4), 14:1–14:29 (2014). https://doi.org/10.1145/2629608
48. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: a library for ω-words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 543–550. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_34
49. 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
50. Lehmann, D., Rabin, M.O.: On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In: White, J., Lipton, R.J., Goldberg, P.C. (eds.) Proceedings of 8th Annual ACM Symposium on Principles of Programming Languages, POPL 1981, Williamsburg, Virginia, USA, January 1981, pp. 133–138. ACM Press (1981). https://doi.org/10.1145/567532.567547
51. Lengál, O., Lin, A.W., Majumdar, R., Rümmer, P.: Fair termination for parameterized probabilistic concurrent systems. In: Legay, A., Margaria, T. (eds.) TACAS 2017, Part I. LNCS, vol. 10205, pp. 499–517. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_29
52. Leroux, J.: Vector addition systems reachability problem (a simpler solution). In: Voronkov, A. (ed.) Proceedings of the Alan Turing Centenary Conference, Turing 100, Manchester, UK, 22–25 June 2012. EPiC Series in Computing, vol. 10, pp. 214–228. EasyChair (2012). https://doi.org/10.29007/bnx2
53. Leroux, J.: Presburger vector addition systems. In: Proceedings of 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, 25–28 June 2013. pp. 23–32. IEEE Computer Society (2013). https://doi.org/10.1109/LICS.2013.7

54. Leroux, J.: Vector addition system reversible reachability problem. Log. Methods Comput. Sci. **9**(1) (2013). https://doi.org/10.2168/LMCS-9(1:5)2013
55. Lin, A.W., Rümmer, P.: Liveness of randomised parameterised systems under arbitrary schedulers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part II. LNCS, vol. 9780, pp. 112–133. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_7
56. Moran, P.A.P.: Random processes in genetics. Math. Proc. Cambridge Philos. Soc. **54**(1), 60–71 (1958). https://doi.org/10.1017/S0305004100033193
57. 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
58. Navlakha, S., Bar-Joseph, Z.: Distributed information processing in biological and computational systems. Commun. ACM **58**(1), 94–102 (2015). https://doi.org/10.1145/2678280
59. Nilsson, M.: Regular model checking. Ph.D. thesis, Uppsala University (2000)
60. Pang, J., Luo, Z., Deng, Y.: On automatic verification of self-stabilizing population protocols. In: Proceedings of 2nd IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE 2008, 17–19 June 2008, Nanjing, China, pp. 185–192. IEEE Computer Society (2008). https://doi.org/10.1109/TASE.2008.8
61. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981). https://doi.org/10.1016/0020-0190(81)90106-X
62. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. Comptes Rendus du I^{er} Congrès des mathématiciens des pays slaves, pp. 192–201 (1929)
63. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_59
64. Szymanski, B.K.: A simple solution to Lamport's concurrent programming problem with linear wait. In: Lenfant, J. (ed.) Proceedings of 2nd International Conference on Supercomputing, ICS 1988, Saint Malo, France, 4–8 July 1988, pp. 621–626. ACM (1988). https://doi.org/10.1145/55364.55425
65. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: Proceedings of 26th Annual Symposium on Foundations of Computer Science, FOCS 1985, Portland, Oregon, USA, 21–23 October 1985, pp. 327–338. IEEE Computer Society (1985). https://doi.org/10.1109/SFCS.1985.12

Stochastic Games with Lexicographic Reachability-Safety Objectives

Krishnendu Chatterjee[1], Joost-Pieter Katoen[3], Maximilian Weininger[2], and Tobias Winkler[3(✉)]

[1] IST Austria, Klosterneuburg, Austria
[2] Technical University of Munich, Munich, Germany
[3] RWTH Aachen University, Aachen, Germany
tobias.winkler@cs.rwth-aachen.de

Abstract. We study turn-based stochastic zero-sum games with lexicographic preferences over reachability and safety objectives. Stochastic games are standard models in control, verification, and synthesis of stochastic reactive systems that exhibit both randomness as well as angelic and demonic non-determinism. Lexicographic order allows to consider multiple objectives with a strict preference order over the satisfaction of the objectives. To the best of our knowledge, stochastic games with lexicographic objectives have not been studied before. We establish determinacy of such games and present strategy and computational complexity results. For strategy complexity, we show that lexicographically optimal strategies exist that are deterministic and memory is only required to remember the already satisfied and violated objectives. For a constant number of objectives, we show that the relevant decision problem is in $\mathsf{NP} \cap \mathsf{coNP}$, matching the current known bound for single objectives; and in general the decision problem is PSPACE-hard and can be solved in $\mathsf{NEXPTIME} \cap \mathsf{coNEXPTIME}$. We present an algorithm that computes the lexicographically optimal strategies via a reduction to computation of optimal strategies in a sequence of single-objectives games. We have implemented our algorithm and report experimental results on various case studies.

1 Introduction

Simple stochastic games (SGs) [26] are zero-sum turn-based stochastic games played over a finite state space by two adversarial players, the Maximizer and Minimizer, along with randomness in the transition function. These games allow the interaction of angelic and demonic non-determinism as well as stochastic uncertainty. They generalize classical models such as Markov decision processes (MDPs) [39] which have only one player and stochastic uncertainty. An objective

This research was funded in part by the TUM IGSSE Grant 10.06 (PARSEC), the German Research Foundation (DFG) project KR 4890/2-1 "Statistical Unbounded Verification", the ERC CoG 863818 (ForM-SMArt), the Vienna Science and Technology Fund (WWTF) Project ICT15-003, and the RTG 2236 UnRAVeL.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 398–420, 2020.
https://doi.org/10.1007/978-3-030-53291-8_21

specifies a desired set of trajectories of the game, and the goal of the Maximizer is to maximize the probability of satisfying the objective against all choices of the Minimizer. The basic decision problem is to determine whether the Maximizer can ensure satisfaction of the objective with a given probability threshold. This problem is among the rare and intriguing combinatorial problems that are $\mathsf{NP} \cap \mathsf{coNP}$, and whether it belongs to P is a major and long-standing open problem. Besides the theoretical interest, SGs are a standard model in control and verification of stochastic reactive systems [4,18,31,39], as well as they provide robust versions of MDPs when precise transition probabilities are not known [22,45].

The multi-objective optimization problem is relevant in the analysis of systems with multiple, potentially conflicting goals, and a trade-off must be considered for the objectives. While the multi-objective optimization has been extensively studied for MDPs with various classes of objectives [1,28,39], the problem is notoriously hard for SGs. Even for multiple reachability objectives, such games are not determined [23] and their decidability is still open.

This work considers SGs with multiple reachability and safety objectives with lexicographic preference order over the objectives. That is, we consider SGs with several objectives where each objective is either reachability or safety, and there is a total preference order over the objectives. The motivation to study such lexicographic objectives is twofold. First, they provide an important special case of general multiple objectives. Second, lexicographic objectives are useful in many scenarios. For example, (i) an autonomus vehicle might have a primary objective to avoid clashes and a secondary objective to optimize performance; and (b) a robot saving lives during fire in a building might have a primary objective to save as many lives as possible, and a secondary objective to minimize energy consumption. Thus studying reactive systems with lexicographic objectives is a very relevant problem which has been considered in many different contexts [7,33]. In particular non-stochastic games with lexicographic objectives [6,25] and MDPs with lexicographic objectives [47] have been considered, but to the best of our knowledge SGs with lexicographic objectives have not been studied.

In this work we present several contributions for SGs with lexicographic reachability and safety objectives. The main contributions are as follows.

- *Determinacy.* In contrast to SGs with multiple objectives that are not determined, we establish determinacy of SGs with lexicographic combination of reachability and safety objectives.
- *Computational complexity.* For the associated decision problem we establish the following: (a) if the number of objectives is constant, then the decision problem lies in $\mathsf{NP} \cap \mathsf{coNP}$, matching the current known bound for SGs with a single objective; (b) in general the decision problem is PSPACE-hard and can be solved in $\mathsf{NEXPTIME} \cap \mathsf{coNEXPTIME}$.
- *Strategy complexity.* We show that lexicographically optimal strategies exist that are deterministic but require finite memory. We also show that memory is only needed in order to remember the already satisfied and violated objectives.

– *Algorithm.* We present an algorithm that computes the unique lexicographic value and the witness lexicographically optimal strategies via a reduction to computation of optimal strategies in a sequence of single-objectives games.
– *Experimental results.* We have implemented the algorithm and present experimental results on several case studies.

Technical Contribution. The key idea is that, given the lexicographic order of the objectives, we can consider them sequentially. After every objective, we remove all actions that are not optimal, thereby forcing all following computation to consider only locally optimal actions. The main complication is that local optimality of actions does not imply global optimality when interleaving reachability and safety, as the latter objective can use locally optimal actions to stay in the safe region without reaching the more important target. We introduce quantified reachability objectives as a means to solve this problem.

Related Work. We present related works on: (a) MDPs with multiple objectives; (b) SGs with multiple objectives; (c) lexicographic objectives in related models; and (d) existing tool support.

(a) MDPs with multiple objectives have been widely studied over a long time [1,39]. In the context of verifying MDPs with multiple objectives, both qualitative objectives such as reachability and LTL [29], as well as quantitative objectives, such as mean payoff [8,13], discounted sum [17], or total reward [34] have been considered. Besides multiple objectives with expectation criterion, other criteria have also been considered, such as, combination with variance [9], or multiple percentile (threshold) queries [8,20,32,41]. Practical applications of MDPs with multiple objectives are described in [2,3,42].
(b) More recently, SGs with multiple objectives have been considered, but the results are more limited [43]. Multiple mean-payoff objectives were first examined in [5] and the qualitative problems are coNP-complete [16]. Some special classes of SGs (namely stopping SGs) have been solved for total-reward objectives [23] and applied to autonomous driving [24]. However, even for the most basic question of solving SGs with multiple reachability objectives, decidability remains open.
(c) The study of lexicographic objectives has been considered in many different contexts [7,33]. Non-stochastic games with lexicographic mean-payoff objectives and parity conditions have been studied in [6] for the synthesis of reactive systems with performance guarantees. Non-stochastic games with multiple ω-regular objectives equipped with a monotonic preorder, which subsumes lexicographic order, have been studied in [12]. Moreover, the beyond worst-case analysis problems studied in [11] also considers primary and secondary objectives, which has a lexicographic flavor. MDPs with lexicographic discounted-sum objectives have been studied in [47], and have been extended with partial-observability in [46]. However, SGs with lexicographic reachability and safety objectives have not been considered so far.

(d) PRISM-Games [37] provides tool support for several multi-player multi-objective settings. MultiGain [10] is limited to generalized mean-payoff MDPs. Storm [27] can, among numerous single-objective problems, solve Markov automata with multiple timed reachability or expected cost objectives [40], multi-cost bounded reachability MDPs [35], and it can provide simple strategies for multiple expected reward objectives in MDPs [28].

Structure of this Paper. After recalling preliminaries and defining the problem in Sect. 2, we first consider games where all target sets are absorbing in Sect. 3. Then, in Sect. 4 we extend our insights to general games, yielding the full algorithm and the theoretical results. Finally, Sect. 5 describes the implementation and experimental evaluation. Section 6 concludes.

2 Preliminaries

Notation. A probability distribution on a finite set A is a function $f : A \to [0, 1]$ such that $\sum_{x \in A} f(x) = 1$. We denote the set of all probability distributions on A by $\mathcal{D}(A)$. Vector-like objects $\boldsymbol{x}$ are denoted in a bold font and we use the notation $\boldsymbol{x}_i$ for the i-th component of $\boldsymbol{x}$. We use $\boldsymbol{x}_{<n}$ as a shorthand for $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n-1})$.

2.1 Basic Definitions

Probabilistic Models. In this paper, we consider *(simple) stochastic games* [26], which are defined as follows. Let $L = \{a, b, \ldots\}$ be a finite set of actions labels.

Definition 1 (SG). *A* stochastic game *(SG) is a tuple* $\mathcal{G} = (S_\Box, S_\Diamond, \mathsf{Act}, P)$ *with* $S := S_\Box \uplus S_\Diamond \neq \emptyset$ *a finite set of states,* $\mathsf{Act} : S \to 2^L \setminus \{\emptyset\}$ *defines finitely many actions available at every state, and* $P : S \times L \to \mathcal{D}(S)$ *is the transition probability function.* $P(s, a)$ *is undefined if* $a \notin \mathsf{Act}(s)$.

We abbreviate $P(s, a)(s')$ to $P(s, a, s')$. We refer to the two players of the game as Max and Min and the sets $S_\Box$ and $S_\Diamond$ are the Max- and Min-states, respectively. As the game is *turn based*, these sets partition the state space S such that in each state it is either Max's or Min's turn. The intuitive semantics of an SG is as follows: In every turn, the corresponding player picks one of the finitely many available actions $a \in \mathsf{Act}(s)$ in the current state s. The game then transitions to the next state according to the probability distribution $P(s, a)$. The winning conditions are not part of the game itself and need to be further specified.

Sinks, Markov Decision Processes and Markov Chains. A state $s \in S$ is called *absorbing* (or sink) if $P(s, a, s) = 1$ for all $a \in \mathsf{Act}(s)$ and $\mathsf{Sinks}(\mathcal{G})$ denotes the set of all absorbing states of SG $\mathcal{G}$. A *Markov Decision Process* (MDP) is an SG where either $S_\Diamond = \emptyset$ or $S_\Box = \emptyset$, i.e. a one-player game. A *Markov Chain* (MC) is an SG where $|\mathsf{Act}(s)| = 1$ for all $s \in S$. For technical reasons, we allow countably infinite state spaces S for both MDPs and MCs.

Strategies. We define the formal semantics of games by means of *paths* and *strategies*. An *infinite path* π is an infinite sequence $\pi = s_0 a_0 s_1 a_1 \cdots \in (S \times L)^\omega$, such that for every $i \in \mathbb{N}$, $a_i \in \mathsf{Act}(s_i)$ and $s_{i+1} \in \{s' \mid P(s_i, a_i, s') > 0\}$. *Finite paths* are defined analogously as elements of $(S \times L)^* \times S$. Note that when considering MCs, every state just has a single action, so an infinite path can be identified with an element of S^ω.

A strategy of player Max is a function $\sigma \colon (S \times L)^* \times S_\Box \to \mathcal{D}(L)$ where $\sigma(\pi s)(s') > 0$ only if $s \in \mathsf{Act}(s)$. It is *memoryless* if $\sigma(\pi s) = \sigma(\pi' s)$ for all $\pi, \pi' \in (S \times L)^*$. More generally, σ has memory of class-size at most m if the set $(S \times L)^*$ can be partitioned in m classes $M_1, \ldots, M_m \subseteq (S \times L)^*$ such that $\sigma(\pi s) = \sigma(\pi' s)$ for all $1 \leq i \leq m$, $\pi, \pi' \in M_i$ and $s \in S_\Box$. A memory of class-size m can be represented with $\lceil \log(m) \rceil$ bits.

A strategy is *deterministic* if $\sigma(\pi s)$ is Dirac for all πs. Strategies that are both memoryless and deterministic are called *MD* and can be identified as functions $\sigma \colon S_\Box \to L$. Notice that there are at most $|L|^{S_\Box}$ different MD strategies, that is, exponentially many in $S_\Box$; in general, there can be uncountably many strategies.

Strategies τ of player Min are defined analogously, with $S_\Box$ replaced by $S_\Diamond$. The set of all strategies of player Max is denoted with Σ_{Max}, the set of all MD strategies with $\Sigma^{\mathsf{MD}}_{\mathsf{Max}}$, and similarly Σ_{Min} and $\Sigma^{\mathsf{MD}}_{\mathsf{Min}}$ for player Min.

Fixing a strategy σ of one player in a game $\mathcal{G}$ yields the *induced MDP* $\mathcal{G}^\sigma$. Fixing a strategy τ of the second player too, yields the *induced MC* $\mathcal{G}^{\sigma,\tau}$. Notice that the induced models are finite if and only if the respective strategies use finite memory.

Given an (induced) MC $\mathcal{G}^{\sigma,\tau}$, we let $\mathbb{P}^{\sigma,\tau}_s$ be its associated probability measure on the Borel-measurable sets of infinite paths obtained from the standard cylinder construction where s is the initial state [39].

Reachability and Safety. In our setting, a *property* is a Borel-measurable set $\Omega \subseteq S^\omega$ of infinite paths in an SG. The *reachability property* $\mathsf{Reach}\ (T)$ where $T \subseteq S$ is the set $\mathsf{Reach}\ (T) = \{s_0 s_1 \ldots \in S^\omega \mid \exists i \geq 0 \colon s_i \in T\}$. The set $\mathsf{Safe}\ (T) = S^\omega \setminus \mathsf{Reach}\ (T)$ is called a *safety property*. Further, for sets $T_1, T_2 \subseteq S$ we define the *until property* $T_1\ \mathsf{U}\ T_2 = \{s_0 s_1 \ldots \in S^\omega \mid \exists i \geq 0 \colon s_i \in T_2 \wedge \forall j < i \colon s_j \in T_1\}$. These properties are measurable (e.g. [4]). A reachability or safety property where the set T satisfies $T \subseteq \mathsf{Sinks}(\mathcal{G})$ is called *absorbing*. For the safety probabilities in an (induced) MC, it holds that $\mathbb{P}_s(\mathsf{Safe}\ (T)) = 1 - \mathbb{P}_s(\mathsf{Reach}\ (T))$. We highlight that an objective $\mathsf{Safe}\ (T)$ is specified by the set of paths to avoid, i.e. paths satisfying the objective remain forever in $S \setminus T$.

2.2 Stochastic Lexicographic Reachability-Safety Games

SGs with lexicographic preferences are a straightforward adaptation of the ideas of e.g. [46] to the game setting. The *lexicographic* order on $\mathbb{R}^n$ is defined as $\boldsymbol{x} \leq_{\mathsf{lex}} \boldsymbol{y}$ iff $\boldsymbol{x}_i \leq \boldsymbol{y}_i$ where $i \leq n$ is the greatest position such that for all $j < i$ it holds that $\boldsymbol{x}_j = \boldsymbol{y}_j$. The position i thus acts like a *tiebreaker*. Notice that for arbitrary sets $X \subseteq [0,1]^n$, suprema and infima exist in the lexicographic order.

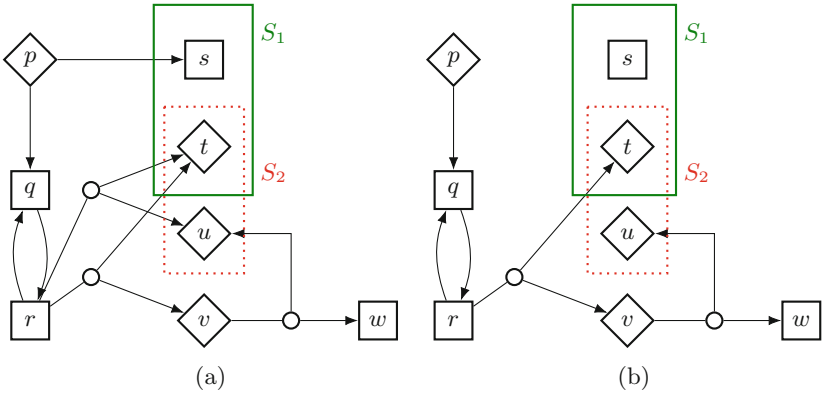


Fig. 1. (a) An example of a stochastic game. Max-states are rendered as squares □ and Min-states as rhombs ◊. Probabilistic choices are indicated with small circles. In this example, all probabilities equal 1/2. The absorbing lex-objective $\boldsymbol{\Omega} = \{\mathsf{Reach}\,(S_1), \mathsf{Safe}\,(S_2)\}$ is indicated by the thick green line around $S_1 = \{s, t\}$ and the dotted red line around $S_2 = \{t, u\}$. Self-loops in sinks are omitted. (b) Restriction of the game to lex-optimal actions only.

Definition 2 (Lex-Objective and Lex-Value). *A* lexicographic reachability-safety objective *(*lex-objective*, for short) is a vector* $\boldsymbol{\Omega} = (\Omega_1, \ldots, \Omega_n)$ *such that* $\Omega_i \in \{\mathsf{Reach}\,(S_i), \mathsf{Safe}\,(S_i)\}$ *with* $S_i \subseteq S$ *for all* $1 \leq i \leq n$*. We call* $\boldsymbol{\Omega}$ absorbing *if all the* Ω_i *are absorbing, i.e., if* $S_i \subseteq \mathsf{Sinks}(\mathcal{G})$ *for all* $1 \leq i \leq n$*. The* lex-(icographic)value *of* $\boldsymbol{\Omega}$ *at state* $s \in S$ *is defined as:*

$$ {}^{\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}(s) = \sup_{\sigma \in \Sigma_{\mathsf{Max}}} \inf_{\tau \in \Sigma_{\mathsf{Min}}} \mathbb{P}_s^{\sigma,\tau}(\boldsymbol{\Omega}) \tag{1} $$

where $\mathbb{P}_s^{\sigma,\tau}(\boldsymbol{\Omega})$ *denotes the* vector $(\mathbb{P}_s^{\sigma,\tau}(\Omega_1), \ldots, \mathbb{P}_s^{\sigma,\tau}(\Omega_n))$ *and the suprema and infima are taken with respect to the order* $\leq_{\mathsf{lex}}$ *on* $[0,1]^n$*.*

Thus the lex-value at state s is the lexicographically supremal vector of probabilities that Max can ensure against all possible behaviors of Min. We will prove in Sect. 4.3 that the supremum and infimum in (1) can be exchanged; this property is called *determinacy*. We omit the superscript $\boldsymbol{\Omega}$ in ${}^{\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}$ if it is clear from the context. We also omit the sets Σ_{Max} and Σ_{Min} in the suprema in (1), e.g. we will just write $\sup_\sigma$.

Example 1 (SGs and lex-values). Consider the SG sketched in Fig. 1a with the lex-objective $\boldsymbol{\Omega} = \{\mathsf{Reach}\,(S_1), \mathsf{Safe}\,(S_2)\}$. Player Max must thus maximize the probability to reach S_1 and, moreover, among all possible strategies that do so, it must choose one that maximizes the probability to avoid S_2 forever. △

Lex-Value of Actions and Lex-Optimal Actions. We extend the notion of value to actions. Let $s \in S$ be a state. The *lex-value of an action* $a \in \mathsf{Act}(s)$ is

defined as $\mathbf{v}^{\mathsf{lex}}(s,a) = \sum_{s'} P(s,a,s')\mathbf{v}^{\mathsf{lex}}(s')$. If $s \in S_{\Box}$, then action a is called *lex-optimal* if $\mathbf{v}^{\mathsf{lex}}(s,a) = \max_{b \in \mathsf{Act}(s)} \mathbf{v}^{\mathsf{lex}}(s,b)$. Lex-optimal actions are defined analogously for states $s \in S_{\Diamond}$ by considering the minimum instead of the maximum. Notice that there is always at least one optimal action because $\mathsf{Act}(s)$ is finite by definition.

Example 2 (Lex-value of actions). We now intuitively explain the lex-values of all states in Fig. 1a. The lex-value of sink states s, t, u and w is determined by their membership in the sets S_1 and S_2. E.g., $\mathbf{v}^{\mathsf{lex}}(s) = (1,1)$, as it is part of the set S_1 that should be reached and not part of the set S_2 that should be avoided. Similarly we get the lex-values of t, u and w as $(1,0)$, $(0,0)$ and $(0,1)$ respectively. State v has a single action that yields $(0,0)$ or $(0,1)$ each with probability $1/2$, thus $\mathbf{v}^{\mathsf{lex}}(v) = (0, 1/2)$.

State p has one action going to s, which would yield $(1,1)$. However, as p is a Min-state, its best strategy is to avoid giving such a high value. Thus, it uses the action going downwards and $\mathbf{v}^{\mathsf{lex}}(p) = \mathbf{v}^{\mathsf{lex}}(q)$. State q only has a single action going to r, so $\mathbf{v}^{\mathsf{lex}}(q) = \mathbf{v}^{\mathsf{lex}}(r)$.

State r has three choices: (i) Going back to q, which results in an infinite loop between q and r, and thus never reaches S_1. So a strategy that commits to this action will not achieve the optimal value. (ii) Going to t or u each with probability $1/2$. In this case, the safety objective is definitely violated, but the reachability objective achieved with $1/2$. (iii) Going to t or v each with probability $1/2$. Similarly to (ii), the probability to reach S_1 is $1/2$, but additionally, there is a $1/2 \cdot 1/2$ chance to avoid S_2. Thus, since r is a Max-state, its lex-optimal choice is the action leading to t or v and we get $\mathbf{v}^{\mathsf{lex}}(r) = (1/2, 1/4)$. △

Notice that with the kind of objectives considered, we can easily swap the roles of Max and Min by exchanging safety objectives with reachability and vice versa. It is thus no loss of generality to consider subsequently introduced notions such as optimal strategies only from the perspective of Max.

Definition 3 (Lex-Optimal Strategies). *A strategy $\sigma \in \Sigma_{\mathsf{Max}}$ is* lex-optimal *for $\boldsymbol{\Omega}$ if for all $s \in S$, $\mathbf{v}^{\mathsf{lex}}(s) = \inf_{\tau'} \mathbb{P}_s^{\sigma,\tau'}(\boldsymbol{\Omega})$. A strategy τ of Min is a* lex-optimal counter-strategy *against σ if $\mathbb{P}_s^{\sigma,\tau}(\boldsymbol{\Omega}) = \inf_{\tau'} \mathbb{P}_s^{\sigma,\tau'}(\boldsymbol{\Omega})$.*

We stress that counter-strategies of Min depend on the strategy chosen by Max.

Locally Lex-Optimal Strategies. An MD strategy σ of Max (Min, resp.) is called *locally lex-optimal* if for all $s \in S_{\Box}$ ($s \in S_{\Diamond}$, resp.) and $a \in \mathsf{Act}(s)$, we have $\sigma(s)(a) > 0$ implies that action a is lex-optimal. Thus, locally lex-optimal strategies only assign positive probability to lex-optimal actions.

Convention. For the rest of the paper, unless stated otherwise, we use $\mathcal{G} = (S_{\Box}, S_{\Diamond}, \mathsf{Act}, P)$ to denote an SG and $\boldsymbol{\Omega} = (\Omega_1, \ldots, \Omega_n)$ is a suitable (not necessarily absorbing) lex-objective, that is $\Omega_i \in \{\mathsf{Reach}\,(S_i), \mathsf{Safe}\,(S_i)\}$ with $S_i \subseteq S$ for all $1 \leq i \leq n$.

3 Lexicographic SGs with Absorbing Targets

In this section, we show how to compute the lexicographic value for SGs where *all target sets are absorbing*. We first show various theoretical results in Sect. 3.1 upon which the algorithm for computing the values and optimal strategies presented in Sect. 3.2 is then built. The main technical difficulty arises from interleaving reachability and safety objectives. In Sect. 4, we will reduce solving general (not necessarily absorbing) SGs to the case with absorbing targets.

3.1 Characterizing Optimal Strategies

This first subsection derives a characterization of lex-optimal strategies in terms of local optimality and an additional reachability condition (Lemma 2 further below). It is one of the key ingredients for the correctness of the algorithm presented later and also gives rise to a (non-constructive) proof of existence of MD lex-optimal strategies in the absorbing case.

We begin with the following lemma that summarizes some straightforward facts we will frequently use. Recall that a strategy is *locally lex-optimal* if it only selects actions with optimal lex-value.

Lemma 1. *The following statements hold for any absorbing lex-objective* $\boldsymbol{\Omega}$*:*

(a) If $\sigma \in \Sigma^{\mathsf{MD}}_{\mathsf{Max}}$ *is lex-optimal and* $\tau \in \Sigma^{\mathsf{MD}}_{\mathsf{Min}}$ *is a lex-optimal counter strategy against* σ*, then* σ *and* τ *are both* locally *lex-optimal. (We do not yet claim that such strategies* σ, τ *always exist.)*

(b) Let $\widetilde{\mathcal{G}}$ *be obtained from* $\mathcal{G}$ *by removing all actions (of both players) that are not locally lex-optimal. Let* $\widetilde{\mathbf{v}}^{\mathsf{lex}}$ *be the lex-values in* $\widetilde{\mathcal{G}}$*. Then* $\widetilde{\mathbf{v}}^{\mathsf{lex}} = \mathbf{v}^{\mathsf{lex}}$*.*

Proof (Sketch). Both claims follow from the definitions of lex-value and lex-optimal strategy. For (b) in particular, we show that a strategy using actions which are not lex-optimal can be transformed into a strategy that achieves a greater (lower, resp.) value. Thus removing the non lex-optimal actions does not affect the lex-value. See [19, Appendix A.1] for more technical details. □

Example 3 (Modified game $\widetilde{\mathcal{G}}$*).* Consider again the SG from Fig. 1a. Recall the lex-values from Example 1. Now we remove the actions that are not locally lex-optimal. This means we drop the action that leads from p to s and the action that leads from r to t or u (Fig. 1b). Since these actions were not used by the lex-optimal strategies, the value in the modified SG is the same as that of the original game. △

Example 4 (Locally lex-optimal does not imply globally lex-optimal). Note that we do not drop the action that leads from r to q, because $\mathbf{v}^{\mathsf{lex}}(r) = \mathbf{v}^{\mathsf{lex}}(q)$, so this action is locally lex-optimal. In fact, a lex-optimal strategy can use it arbitrarily many times without reducing the lex-value, as long as eventually it picks the action leading to t or v. However, if we only played the action leading to q, the

lex-value would be reduced to $(0, 1)$ as we would not reach S_1, but would also avoid S_2.

We stress the following consequence of this: Playing a locally lex-optimal strategy is not necessarily globally lex-optimal. It is not sufficient to just restrict the game to locally lex-optimal actions of the previous objectives and then solve the current one. Note that in fact the optimal strategy for the second objective $\mathsf{Safe}\ (S_2)$ would be to remain in $\{p, q\}$; however, we must not pick this safety strategy, before we have not "tried everything" for all previous reachability objectives, in this case reaching S_1. △

This idea of "trying everything" for an objective $\mathsf{Reach}\ (S_i)$ is equivalent to the following: either reach the target set S_i, or reach a set of states from which S_i cannot be reached anymore. Formally, let $\mathsf{Zero}_i = \{s \in S \mid \mathbf{v}_i^{\mathsf{lex}}(s) = 0\}$ be the set of states that cannot reach the target set S_i anymore. Note that it depends on the lex-value, not the single-objective value. This is important, as the single-objective value could be greater than 0, but a more important objective has to be sacrificed to achieve it.

We define the set of states where we have "tried everything" for all reachability objectives as follows:

Definition 4 (Final Set). *For absorbing $\boldsymbol{\Omega}$, let $R_{<i} = \{j < i \mid \Omega_j = \mathsf{Reach}\ (S_j)\}$. We define the* final set $F_{<i} = \bigcup_{k \in R_{<i}} S_k \cup \bigcap_{k \in R_{<i}} \mathsf{Zero}_k$ *with the convention that $F_{<i} = S$ if $R_{<i} = \emptyset$. We also let $F = F_{<n+1}$.*

The final set contains all target states as well as the states that have lex-value 0 for all reachability objectives; we need the intersection of the sets Zero_k, because as long as a state still has a positive probability to reach any target set, its optimal behaviour is to try that.

Example 5 (Final set). For the game in Fig. 1, we have $\mathsf{Zero}_1 = \{u, v, w\}$ and thus $F = \mathsf{Zero}_1 \cup S_1 = \{s, t, u, v, w\}$. An MD lex-optimal strategy of Max must almost-surely reach this set against any strategy of Min; only then it has "tried everything". △

The following lemma characterizes MD lex-optimal strategies in terms of local lex-optimality and the final set.

Lemma 2. *Let $\boldsymbol{\Omega}$ be an absorbing lex-objective and $\sigma \in \Sigma^{\mathsf{MD}}_{\mathsf{Max}}$. Then σ is lex-optimal for $\boldsymbol{\Omega}$ if and only if σ is locally lex-optimal and for all $s \in S$ we have*

$$\forall \tau \in \Sigma^{\mathsf{MD}}_{\mathsf{Min}} \colon \mathbb{P}_s^{\sigma,\tau}(\mathsf{Reach}\ (F)) = 1. \qquad (\star)$$

Proof (Sketch). The "*if*"-direction is shown by induction on the number n of targets. We make a case distinction according to the type of Ω_n: If it is safety, then we prove that local lex-optimality is already sufficient for global lex-optimality. Else if Ω_n is reachability, then intuitively, the additional condition $(\star)$ ensures that the strategy σ indeed "tries everything" and either reaches the target S_n or

eventually a state in Zero_n where the opponent Min can make sure that Max cannot escape. The technical details of these assertions rely on a fixpoint characterization of the reachability probabilities combined with the classic Knaster-Tarski Fixpoint Theorem [44] and are given in [19, Appendix A.2].

For the "*only if*"-direction recall that lex-optimal strategies are necessarily locally lex-optimal by Lemma 1 (a). Further let i be such that $\Omega_i = \mathsf{Reach}\ (S_i)$ and assume for contradiction that σ remains forever within $S \setminus (S_i \cup \mathsf{Zero}_i)$ with positive probability against some strategy of Min. But then σ visits states with positive lex-value for Ω_i infinitely often without ever reaching S_i. Thus σ is not lex-optimal, contradiction. □

Finally, this characterization allows us to prove that MD lex-optimal strategies exist for absorbing objectives.

Theorem 1. *For an absorbing lex-objective $\boldsymbol{\Omega}$, there exist MD lex-optimal strategies for both players.*

Proof (Sketch). We consider the subgame $\widetilde{\mathcal{G}}$ obtained by removing lex-suboptimal actions for both players and then show that the (single-objective) value of $\mathsf{Reach}\ (F)$ in $\widetilde{\mathcal{G}}$ equals 1. An optimal MD strategy for $\mathsf{Reach}\ (F)$ exists [26]; further, it is locally lex-optimal, because we are in $\widetilde{\mathcal{G}}$, and it reaches F almost surely. Thus, it is lex-optimal for $\boldsymbol{\Omega}$ by the "*if*"-direction of Lemma 2. See [19, Appendix A.3] for more details on the proof. □

3.2 Algorithm for SGs with Absorbing Targets

Theorem 1 is not constructive because it relies on the values $\mathbf{v}^{\mathsf{lex}}$ without showing how to compute them. Computing the values and constructing an optimal strategy for Max in the case of an absorbing lex-objective is the topic of this subsection.

Definition 5 (QRO). *A* quantified reachability objective *(QRO) is determined by a function $q\colon S' \to [0,1]$ where $S' \subseteq S$. For all strategies σ and τ, we define:*

$$\mathbb{P}_s^{\sigma,\tau}(\mathsf{Reach}\ (q)) = \sum_{t \in S'} \mathbb{P}_s^{\sigma,\tau}((S \setminus S')\ \mathsf{U}\ t) \cdot q(t).$$

Intuitively, a QRO generalizes its standard Boolean counterpart by additionally assigning a weight to the states in the target set S'. Thus the probability of a QRO is obtained by computing the sum of the $q(t)$, $t \in S'$, weighted by the probability to avoid S' until reaching t. Note that this probability does not depend on what happens after reaching S'; so it is unaffected by making all states in S' absorbing.

In Sect. 4, we need the dual notion of a quantified safety property, defined as $\mathbb{P}_s^{\sigma,\tau}(\mathsf{Safe}\ (q)) = 1 - \mathbb{P}_s^{\sigma,\tau}(\mathsf{Reach}\ (q))$; intuitively, this amounts to minimizing the reachability probability.

Remark 1. A usual reachability property $\mathsf{Reach}\ (S')$ is a special case of a quantified one with $q(s) = 1$ for all $s \in S'$. Vice versa, quantified properties can be easily reduced to usual ones defined only by the set S': Convert all states $t \in S'$ into sinks, then for each such t prepend a new state t' with a single action a and $P(t', a, t) = q(t)$ and $P(t', a, \bot) = 1 - q(t)$ where $\bot$ is a sink state. Finally, redirect all transitions leading into t to t'. Despite this equivalence, it turns out to be convenient and natural to use QROs.

Example 6 (QRO). Example 4 illustrated that solving a safety objective after a reachability objective can lead to problems, as the optimal strategy for $\mathsf{Safe}\ (S_2)$ did not use the action that actually reached S_1. In Example 5 we indicated that the final set $F = \{s, t, u, v, w\}$ has to be reached almost surely, and among those states the ones with the highest safety values should be preferred. This can be encoded in a QRO as follows: Compute the values for the $\mathsf{Safe}\ (S_2)$ objective for the states in F. Then construct the function $q_2 \colon F \to [0, 1]$ that maps all states in F to their safety value, i.e., $q_2 : \{s \mapsto 1, t \mapsto 0, u \mapsto 0, v \mapsto 1/2, w \mapsto 1\}$. △

Thus using QROs, we can effectively reduce (interleaved) safety objectives to quantified *reachability* objectives:

Lemma 3 (Reduction Safe → Reach). *Let $\boldsymbol{\Omega}$ be an absorbing lex-objective with $\Omega_n = \mathsf{Safe}\ (S_n)$, $q_n \colon F \to [0, 1]$ with $q_n(t) = \mathbf{v}_n^{\mathsf{lex}}(t)$ for all $t \in F$ where F is the final set (Definition 4), and $\boldsymbol{\Omega}' = (\Omega_1, \ldots, \Omega_{n-1}, \mathsf{Reach}\ (q_n))$. Then: ${}^{\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}} = {}^{\boldsymbol{\Omega}'}\mathbf{v}^{\mathsf{lex}}$.*

Proof (Sketch). By definition, ${}^{\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}(s) = {}^{\boldsymbol{\Omega}'}\mathbf{v}^{\mathsf{lex}}(s)$ for all $s \in F$, so we only need to consider the states in $S \setminus F$. Since any lex-optimal strategy for $\boldsymbol{\Omega}$ or $\boldsymbol{\Omega}'$ must also be lex-optimal for $\boldsymbol{\Omega}_{<n}$, we know by Lemma 2 that such a strategy reaches $F_{<n}$ almost-surely. Note that we have $F_{<n} = F$, as the n-th objective, either the QRO or the safety objective, does not add any new states to F. The reachability objective $\mathsf{Reach}\ (q_n)$ weighs the states in F with their lexicographic safety values $\mathbf{v}_n^{\mathsf{lex}}$. Thus we additionally ensure that in order to reach F, we use those actions that give us the best safety probability afterwards. In this way we obtain the correct lex-values $\mathbf{v}_n^{\mathsf{lex}}$ even for states in $S \setminus F$. See [19, Appendix A.4] for the full technical proof. □

Example 7 (Reduction Safe → Reach). Recall Example 6. By the preceding Lemma 3, computing $\sup_\sigma \inf_\tau \mathbb{P}_s^{\sigma,\tau}(\mathsf{Reach}\ (S_1), \mathsf{Reach}\ (q_2))$ yields the correct lex-value $\mathbf{v}^{\mathsf{lex}}(s)$ for all $s \in S$. Consider for instance state r in the running example: The action leading to q is clearly suboptimal for $\mathsf{Reach}\ (q_2)$ as it does not reach F. Both other actions surely reach F. However, since $q_2(t) = q_2(u) = 0$ while $q_2(v) = 1/2$, the action leading to u and v is preferred over that leading to t and u, as it ensures the higher safety probability after reaching F. △

We now explain the basic structure of Algorithm 1. More technical details are explained in the proof sketch of Theorem 2 and the full proof is in [19, Appendix A.5]. The idea of Algorithm 1 is, as sketched in Sect. 3.1, to consider the objectives sequentially in the order of importance, i.e., starting with Ω_1.

Algorithm 1. Solve absorbing lex-objective

Input: SG $\mathcal{G}$, absorbing lex-objective $\boldsymbol{\Omega} = (\Omega_1, \ldots, \Omega_n)$
Output: Vector of lex-values $\mathbf{v}^{\mathsf{lex}}$, MD lex-optimal strategy σ for Max

1: **procedure** `SolveAbsorbing`$(\mathcal{G}, \boldsymbol{\Omega})$
2: initialize $\mathbf{v}^{\mathsf{lex}}$ and σ arbitrarily
3: $\widetilde{\mathcal{G}} \leftarrow \mathcal{G}$ ▷ Consider whole game in the beginning.
4: **for** $1 \leq i \leq n$ **do**
5: $(v, \widetilde{\sigma}) \leftarrow$ `SolveSingleObj`$(\widetilde{\mathcal{G}}, \Omega_i)$
6: **if** $\Omega_i = \mathsf{Safe}\ (S_i)$ **then**
7: $F_{<i} \leftarrow$ final set with respect to $\widetilde{\mathcal{G}}$ and $\boldsymbol{\Omega}_{<i}$ ▷ see Def. 4
8: $q_i(s) \leftarrow v(s)$ for all $s \in F_{<i}$ ▷ see Def. 5
9: $(v, \sigma_Q) \leftarrow$ `SolveSingleObj`$(\widetilde{\mathcal{G}}, \mathsf{Reach}\ (q_i))$
10: **end if**
11: $\widetilde{\mathcal{G}} \leftarrow$ restriction of $\widetilde{\mathcal{G}}$ to optimal actions w.r.t. v
12: $\mathbf{v}_i^{\mathsf{lex}} \leftarrow v$
13: **for** $s \in S$ **do**
14: **if** $(\Omega_i = \mathsf{Reach}\ (S_i)$ and $v(s) > 0)$ or $(\Omega_i = \mathsf{Safe}\ (S_i)$ and $s \in F_{<i})$ **then**
15: $\sigma(s) \leftarrow \widetilde{\sigma}(s)$ ▷ Strategy improvement
16: **else if** $\Omega_i = \mathsf{Safe}\ (S_i)$ and $s \notin F_{<i}$
17: $\sigma(s) \leftarrow \sigma_Q(s)$
18: **end if**
19: **end for**
20: **end for**
return $(\mathbf{v}^{\mathsf{lex}}, \sigma)$
21: **end procedure**

The i-th objective is solved (Lines 5–10) and the game is restricted to only the locally optimal actions (Line 11). This way, in the i-th iteration of the main loop, only actions that are locally lex-optimal for objectives 1 through $(i-1)$ are considered. Finally, we construct the optimal strategy and update the result variables (Lines 12–19).

Theorem 2. *Given an SG $\mathcal{G}$ and an absorbing lex-objective $\boldsymbol{\Omega} = (\Omega_1, \ldots, \Omega_n)$, Algorithm 1 correctly computes the vector of lex-values $\mathbf{v}^{\mathsf{lex}}$ and an MD lex-optimal strategy σ for player* Max*. It needs n calls to a single objective solver.*

Proof (Sketch).

- **$\widetilde{\mathcal{G}}$-invariant:** For $i > 1$, in the i-th iteration of the loop, $\widetilde{\mathcal{G}}$ is the original SG restricted to only those actions that are locally lex-optimal for the targets 1 to $(i-1)$; this is the case because Line 11 was executed for all previous targets.
- **Single-objective case:** The single-objective that is solved in Line 5 can be either reachability or safety. We can use any (precise) single-objective solver as a black box, e.g. strategy iteration [36]. Recall that by Remark 1, it is no problem to call a single-objective solver with a QRO since there is a trivial reduction.
- **QRO for safety:** If an objective is of type reachability, no further steps need to be taken; if on the other hand it is safety, we need to ensure that the problem explained in Example 4 does not occur. Thus we compute the final set $F_{<i}$ for the i-th target and then construct and solve the QRO as in Lemma 3.

– **Resulting strategy:** When storing the resulting strategy, we again need to avoid errors induced by the fact that locally lex-optimal actions need not be globally lex-optimal. This is why for a reachability objective, we only update the strategy in states that have a positive value for the current objective; if the value is 0, the current strategy does not have any preference, and we need to keep the old strategy. For safety objectives, we need to update the strategy in two ways: for all states in the final set $F_{<i}$, we set it to the safety strategy $\widetilde{\sigma}$ (from Line 5) as within $F_{<i}$ we do not have to consider the previous reachability objectives and therefore must follow an optimal safety strategy. For all states in $S \setminus F_{<i}$, we set it to the reachability strategy from the QRO σ_Q (from Line9). This is correct, as σ_Q ensures almost-sure reachability of $F_{<i}$ which is necessary to satisfy all preceding reachability objectives; moreover σ_Q prefers those states in $F_{<i}$ that have a higher safety value (cf. Lemma 3).
– **Termination:** The main loop of the algorithm invokes `SolveSingleObj` for each of the n objectives. □

4 General Lexicographic SGs

We now consider $\boldsymbol{\Omega}$ where $S_i \subseteq \mathsf{Sinks}(\mathcal{G})$ does *not* necessarily hold. Section 4.1 describes how we can reduce these general lex-objectives to the absorbing case. The resulting algorithm is given in Sect. 4.2 and the theoretical implications in Sect. 4.3.

4.1 Reducing General Lexicographic SGs to SGs with Absorbing Targets

In general lexicographic SG, strategies need memory, because they need to remember which of the S_i have already been visited and behave accordingly. We formalize the solution of such games by means of *stages*. Intuitively, one can think of a stage as a copy of the game with less objectives, or as the sub-game that is played after visiting some previously unseen set S_i.

Definition 6 (Stage). *Given an arbitrary lex-objective $\boldsymbol{\Omega} = (\boldsymbol{\Omega}_1, \dots, \boldsymbol{\Omega}_n)$ and a set $I \subseteq \{i \leq n\}$, a* stage $\boldsymbol{\Omega}(I)$ *is the objective vector where the objectives $\boldsymbol{\Omega}_i$ are removed for all $i \in I$.*

For state $s \in S$, let $\boldsymbol{\Omega}(s) = \boldsymbol{\Omega}(\{i \mid s \in S_i\})$. If a stage contains only one objective, we call it simple.

Example 8 (Stages). Consider the SG in Fig. 2a. As there are two objectives, there are four possible stages: The one where we consider both objectives (the region denoted with $\boldsymbol{\Omega}$ in Fig. 2b), the *simple* ones where we consider only one of the objectives (regions $\boldsymbol{\Omega}(\{1\})$ and $\boldsymbol{\Omega}(\{2\})$), and the one where both objectives have been visited. The last stage is trivial since there are no more objectives, hence we do not depict it and do not have to consider it. The actions of q and r are omitted in the $\boldsymbol{\Omega}$-stage, as upon visiting these states, a new stage begins.

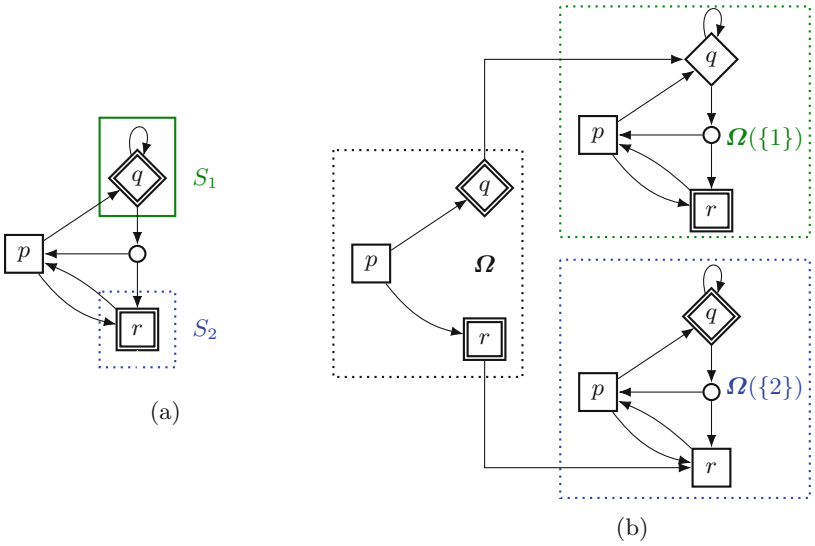


Fig. 2. (a) SG with non-absorbing lex-objective $\boldsymbol{\Omega} = (\mathsf{Reach}\ (S_1), \mathsf{Reach}\ (S_2))$. (b) The three stages identified by the sub-objectives $\boldsymbol{\Omega}$, $\boldsymbol{\Omega}(\{1\}) = (\mathsf{Reach}\ (S_2))$ and $\boldsymbol{\Omega}(\{2\}) = (\mathsf{Reach}\ (S_1))$. The two stages on the right are both *simple*.

Consider the simple stages: in stage $\boldsymbol{\Omega}(\{1\})$, q has value 0, as it is a Min-state and will use the self-loop to avoid reaching $r \in S_2$. In stage $\boldsymbol{\Omega}(\{2\})$, both p and r have value 1, as they can just go to the target state $q \in S_1$. Combining this knowledge, we can get an optimal strategy for every state. In particular, note that an optimal strategy for state p needs memory: First go to r and thereby reach stage $\boldsymbol{\Omega}(\{2\})$. Afterwards, go from r to p and now, on the second visit in a different stage, use the other action in p to reach q. In this example, we observe another interesting fact about lexicographic games: it can be optimal to first satisfy less important objectives. △

In the example, we combined our knowledge of the sub-stages to find the lex-values for the whole lex-objective. In general, the values for the stages are numbers in $[0, 1]$. Thus we reuse the idea of *quantified* reachability and safety objectives, see Definition 5.

For all $1 \leq i \leq n$, let $q_i \colon \bigcup_{j \leq n} S_j \to [0, 1]$ by defined by:

$$q_i(s) = \begin{cases} 1 & \text{if } s \in S_i \text{ and else:} \\ {}^{\boldsymbol{\Omega}(s)}\mathbf{v}_i^{\mathsf{lex}}(s) & \text{if } \Omega_i \text{ is reachability} \\ 1 - {}^{\boldsymbol{\Omega}(s)}\mathbf{v}_i^{\mathsf{lex}}(s) & \text{if } \Omega_i \text{ is safety.} \end{cases}$$

To keep the correct type of every objective, we let $\mathsf{q}\boldsymbol{\Omega} = (\mathsf{type}_1(q_1), \ldots, \mathsf{type}_n(q_n))$ where for all $1 \leq i \leq n$, $\mathsf{type}_i = \mathsf{Reach}$ if $\Omega_i = \mathsf{Reach}\ (S_i)$ and else $\mathsf{type}_i = \mathsf{Safe}$ if

$\Omega_i = \mathsf{Safe}\ (S_i)$. So we have now reduced a general lexicographic objective $\boldsymbol{\Omega}$ to a vector of quantitative objectives $\mathtt{q}\boldsymbol{\Omega}$. Lemma 4 shows that this reduction preserves the values.

Lemma 4. *For arbitrary lex-objectives $\boldsymbol{\Omega}$ it holds that ${}^{\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}} = {}^{\mathtt{q}\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}$.*

Proof (Sketch). We write $\mathfrak{S} = \bigcup_{j \le n} S_j$ for the sake of readability in this sketch. By induction on the length n of the lex-objective $\boldsymbol{\Omega}$, it is easy to show that the equation holds in states $s \in \mathfrak{S}$, i.e., ${}^{\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}(s) = {}^{\mathtt{q}\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}(s)$. For a state s which is not contained in any of the S_j, and for any strategies σ, τ we have the following equation

$$\mathbb{P}_s^{\sigma,\tau}(\mathsf{Reach}\ (S_i)) = \sum_{\pi t \in Paths_{fin}(\mathfrak{S})} \mathbb{P}_s^{\sigma,\tau}(\pi t) \cdot \mathbb{P}_{\pi t}^{\sigma,\tau}(\mathsf{Reach}\ (S_i))$$

where $Paths_{fin}(\mathfrak{S}) = \{\pi t \in ((S \setminus \mathfrak{S}) \times L)^* \times S \mid t \in \mathfrak{S}\}$ denotes the set of all finite paths to a state in $\mathfrak{S}$ in the Markov chain $\mathcal{G}^{\sigma,\tau}$ and $\mathbb{P}_s^{\sigma,\tau}(\pi t)$ is the probability of such a path when $\mathcal{G}^{\sigma,\tau}$ starts in s. From this we deduce that in order to maximize the left hand size of the equation in the lexicographic order, we should play such that we prefer reaching states in $\mathfrak{S}$ where q_i has a higher value; that is, we should maximize the QRO $\mathsf{Reach}\ (q_i)$. The argument for safety is similar and detailed in [19, Appendix A.6]. □

The functions q_i involved in $\mathtt{q}\boldsymbol{\Omega}$ *all have the same domain* $\bigcup_{j \le n} S_j$. Hence we can, as mentioned below Definition 5, consider $\mathtt{q}\boldsymbol{\Omega}$ on the game where all states in $\bigcup_{j \le n} S_j$ are sinks without changing the lex-value. This is precisely the definition of an absorbing game, and hence we can compute ${}^{\mathtt{q}\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}$ using Algorithm 1 from Sect. 3.2.

4.2 Algorithm for General SG

Algorithm 2 computes the lex-value ${}^{\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}$ for a given lexicographic objective $\boldsymbol{\Omega}$ and an arbitrary SG $\mathcal{G}$. We highlight the following technical details:

- **Reduction to absorbing case:** We just have seen, that once we have the quantitative objective vector $\mathtt{q}\boldsymbol{\Omega}$, we can use the algorithm for absorbing SG (Line 12).
- **Computing the quantitative objective vector:** To compute $\mathtt{q}\boldsymbol{\Omega}$, the algorithm calls itself recursively on all states in the union of all target sets (Line 5–7). We annotated this recursive call "With dynamic programming", as we can reuse the results of the computations. In the worst case, we have to solve all $2^n - 1$ possible non-empty stages. Finally, given the values ${}^{\boldsymbol{\Omega}(s)}\mathbf{v}^{\mathsf{lex}}$ for all $s \in \bigcup_{j \le n} S_j$, we can construct the quantitative objective (Line 9 and 11) that is used for the call to `SolveAbsorbing`.
- **Termination:** Since there are finitely many objectives in $\boldsymbol{\Omega}$ and in every recursive call at least one objective is removed from consideration, eventually we have a *simple* objective that can be solved by `SolveSingleObj` (Line 3).

Algorithm 2. Solve general lex-objective

Input: SG $\mathcal{G}$, lex-objective $\boldsymbol{\Omega} = (\Omega_1, \ldots, \Omega_n)$
Output: Lex-values ${}^{\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}$, lex-optimal $\sigma \in \Sigma_{\mathsf{Max}}$ with memory of class-size $\leq 2^n - 1$

1: **procedure** `SolveLex`$(\mathcal{G}, \boldsymbol{\Omega})$
2: **if** $\boldsymbol{\Omega}$ is *simple* **then**
3: **return** `SolveSingleObj`$(\mathcal{G}, \Omega_1)$
4: **end if**
5: **for** $s \in \bigcup_{j \leq n} S_j$ **do**
6: $\left({}^{\boldsymbol{\Omega}(s)}\mathbf{v}^{\mathsf{lex}},\ {}^{\boldsymbol{\Omega}(s)}\sigma\right) \leftarrow$ `SolveLex`$(\mathcal{G}, \boldsymbol{\Omega}(s))$ ▷ With dynamic programming
7: **end for**
8: **for** $1 \leq i \leq n$ **do**
9: Let $q_i \colon \bigcup_{j \leq n} S_j \to [0,1]$, $q_i(s) \leftarrow \begin{cases} 1 & \text{if } s \in S_i \text{ and else:} \\ {}^{\boldsymbol{\Omega}(s)}\mathbf{v}_i^{\mathsf{lex}}(s) & \text{if } \mathsf{type}(\Omega_i) = \mathsf{Reach} \\ 1 - {}^{\boldsymbol{\Omega}(s)}\mathbf{v}_i^{\mathsf{lex}}(s) & \text{if } \mathsf{type}(\Omega_i) = \mathsf{Safe} \end{cases}$
10: **end for**
11: $\mathtt{q}\boldsymbol{\Omega} \leftarrow (\mathsf{type}_1(q_1), \ldots, \mathsf{type}_n(q_n))$
12: $({}^{\mathtt{q}\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}},\ {}^{\mathtt{q}\boldsymbol{\Omega}}\sigma) \leftarrow$ `SolveAbsorbing`$(\mathcal{G}, \mathtt{q}\boldsymbol{\Omega})$
13: $\sigma \leftarrow$ adhere to ${}^{\mathtt{q}\boldsymbol{\Omega}}\sigma$ until some $s \in \bigcup_{j \leq n} S_j$ is reached. Then adhere to ${}^{\boldsymbol{\Omega}(s)}\sigma$.
14: **return** $({}^{\mathtt{q}\boldsymbol{\Omega}}\mathbf{v}^{\mathsf{lex}}, \sigma)$
15: **end procedure**

– **Resulting strategy:** The resulting strategy is composed in Line 13: It adheres to the strategy for the quantitative query ${}^{\mathtt{q}\boldsymbol{\Omega}}\sigma$ until some $s \in \bigcup_{j \leq n} S_j$ is reached. Then, to achieve the values promised by $q_i(s)$ for all i with $s \notin S_i$, it adheres to ${}^{\boldsymbol{\Omega}(s)}\sigma$, the optimal strategy for stage $\boldsymbol{\Omega}(s)$ obtained by the recursive call.

Corollary 1. *Given an SG $\mathcal{G}$ and an arbitrary lex-objective $\boldsymbol{\Omega} = (\Omega_1, \ldots, \Omega_n)$, Algorithm 2 correctly computes the vector of lex-values $\mathbf{v}^{\mathsf{lex}}$ and a deterministic lex-optimal strategy σ of player* Max *which uses memory of class-size $\leq 2^n - 1$. The algorithm needs at most $2^n - 1$ calls to* `SolveAbsorbing` *or* `SolveSingleObj`.

Proof. Correctness of the algorithm and termination follows from the discussion of the algorithm, Lemma 4 and Theorem 2. □

4.3 Theoretical Implications: Determinacy and Complexity

Theorem 3 below states that lexicographic games are *determined* for arbitrary lex-objectives $\boldsymbol{\Omega}$. Intuitively, this means that the lex-value is independent from the player who fixes their strategy first. Recall that this property does not hold for non-lexicographic multi-reachability/safety objectives [23].

Theorem 3 (Determinacy). *For general SG $\mathcal{G}$ and lex-objective $\boldsymbol{\Omega}$, it holds for all $s \in S$ that:*

$$\mathbf{v}^{\mathsf{lex}}(s) = \sup_{\sigma} \inf_{\tau} \mathbb{P}_s^{\sigma,\tau}(\boldsymbol{\Omega}) = \inf_{\tau} \sup_{\sigma} \mathbb{P}_s^{\sigma,\tau}(\boldsymbol{\Omega}).$$

Proof. This statement follows because single-objective games are determined [26] and Algorithm 2 obtains all values by either solving single-objective instances directly (Line 3) or calling Algorithm 1, which also reduces everything to the single-objective case (Line 5 of Algorithm 1). Thus the sup-inf values $\mathbf{v}^{\mathsf{lex}}$ returned by the algorithm are in fact equal to the inf-sup values. □

By analyzing Algorithm 2, we also get the following complexity results:

Theorem 4 (Complexity). *For any SG $\mathcal{G}$ and lex-objective $\boldsymbol{\Omega} = (\Omega_1, \ldots, \Omega_n)$:*

1. Strategy complexity: *Deterministic strategies with $2^n - 1$ memory-classes (i.e., bit-size n) are sufficient and necessary for lex-optimal strategies.*
2. Computational complexity: *The lex-game decision problem ($\mathbf{v}^{\mathsf{lex}}(s_0) \geq_{\mathsf{lex}} \boldsymbol{x}$?) is* PSPACE*-hard and can be solved in* NEXPTIME ∩ coNEXPTIME*. If n is a constant or $\boldsymbol{\Omega}$ is absorbing, then it is contained in* NP ∩ coNP*.*

Proof. 1. For each stage, Algorithm 2 computes an MD strategy for the quantitative objective. These strategies are then concatenated whenever a new stage is entered. Equivalently, every stage has an MD strategy for every state, so as there are at most $2^n - 1$ stages (since there are n objectives), the strategy needs at most $2^n - 1$ states of memory; these can be represented with n bits. Intuitively, we save for every target set whether it has been visited. The memory lower bound already holds in non-stochastic reachability games where all n targets have to be visited with certainty [30].

2. The work of [41] shows that in MDPs, it is PSPACE-hard to decide if n targets can be visited almost-surely. This problem trivially reduces to ours. For the NP upper bound, observe that there are at most $2^n - 1$ stages, i.e., a constant amount if n is assumed to be constant (or even just one stage if $\boldsymbol{\Omega}$ is absorbing). Thus we can guess an MD strategy for player Max in every stage. The guessed overall strategy can then be checked by analyzing the induced MDP in polynomial time [29]. The same procedure works for player Min and since the game is determined, we have membership in coNP. In the same way we obtain the NEXPTIME ∩ coNEXPTIME upper bound in the general case where n is arbitrary. □

We leave the question whether PSPACE is also an upper bound open. The main obstacle towards proving PSPACE-membership is that it is unclear if the lex-value – being dependent on the value of *exponentially* many stages in the worst-case – may actually have exponential bit-complexity.

5 Experimental Evaluation

In this section, we report the results of a series of experiments made with a prototypical implementation of our algorithm.

Case Studies. We have considered the following case studies for our experiments:

Dice. This example is shipped with PRISM-games [37] and models a simple dice game between two players. The number of throws in this game is a configurable parameter, which we instantiate with 10, 20 and 50. The game has three possible outcomes: Player Max wins, Player Min wins or draw. A natural lex-objective is thus to maximize the winning probability and then the probability of a draw.

Charlton. This case study [24] is also included in PRISM-games. It models an autonomous car navigating through a road network. A natural lex-objective is to minimize the probability of an accident (possibly damaging human life) and then maximize the probability to reach the destination.

Hallway (HW). This instance is based on the Hallway example standard in the AI literature [15,38]. A robot can move north, east, south or west in a known environment, but each move only succeeds with a certain probability and otherwise rotates or moves the robot in an undesired direction. We extend the example by a target wandering around based on a mixture of probabilistic and demonic non-deterministic behavior, thereby obtaining a stochastic game modeling for instance a panicking human in a building on fire. Moreover, we assume a 0.01 probability of damaging the robot when executing certain movements; the damaged robot's actions succeed with even smaller probability. The primary objective is to save the human and the secondary objective is to avoid damaging the robot. We use square grid-worlds of sizes 5×5, 8×8 and 10×10.

Avoid the Observer (AV). This case study is inspired by a similar example in [14]. It models a game between an intruder and an observer in a grid-world. The grid can have different sizes as in HW, and we use 10×10, 15×15 and 20×20. The most important objective of the intruder is to avoid the observer, its secondary objective is to exit the grid. We assume that the observer can only detect the intruder within a certain distance and otherwise makes random moves. At every position, the intruder moreover has the option to stay and search to find a precious item. In our example, this occurs with probability 0.1 and is assumed to be the third objective.

Implementation and Experimental Results. We have implemented our algorithm within PRISM-games [37]. Since PRISM-games does not provide an *exact* algorithm to solve SGs, we used the available value iteration to implement our single-objective blackbox. Note that since this value iteration is not exact for single-objective SGs, we cannot compute the exact lex-values. Nevertheless, we can still measure the overhead introduced by our algorithm compared to a single-objective solver.

In our implementation, value iteration stops if the values do not change by more than 10^{-8} per iteration, which is PRISM's default configuration. The experiments were conducted on a 2.4 GHz Quad-Core Intel© Core™ i5 processor, with 4 GB of RAM available to the Java VM. The results are reported in Table 1. We only recorded the run time of the actual algorithms; the time

needed to parse and build the model is excluded. All numbers are rounded to full seconds. All instances (even those with state spaces of order 10^6) could be solved within a few minutes.

Table 1. Experimental Results. The two leftmost columns of the table show the type of the lex-objective, the name of the case studies, possibly with scaling parameters, and the number of states in the model. The next three columns give the verification times (excluding time to parse and build the model), rounded to full seconds. The final three columns provide the average number of actions for the original SG as well as all considered subgames $\widetilde{\mathcal{G}}$ in the main stage, and lastly the fraction of stages considered, i.e. the stages solved by the algorithm compared to the theoretically maximal possible number of stages $(2^n - 1)$.

Model	$\lvert S\rvert$	Time			Avg. actions		Stages
		Lex.	First	All	$\mathcal{G}$	$\widetilde{\mathcal{G}}$	
R − R							
Dice[10]	4,855	<1	<1	<1	1.42	1.41	1/3
Dice[20]	16,915	<1	<1	<1	1.45	1.45	1/3
Dice[50]	96,295	3	2	2	1.48	1.48	1/3
S − R							
Charlton	502	<1	<1	<1	1.56	1.07	3/3
R − S							
HW[5 × 5]	25,000	10	7.15	7	2.44	1.02	3/3
HW[8 × 8]	163,840	152	117	117	2.50	1.01	3/3
HW[10 × 10]	400,000	548	435	435	2.52	1.01	3/3
S–R–R							
AV[10 × 10]	106,524	15	<1	10	2.17	1.55, 1.36	4/7
AV[15 × 15]	480,464	85	<1	50	2.14	1.52, 1.36	4/7
AV[20 × 20]	1,436,404	281	3	172	2.13	1.51, 1.37	4/7

The case studies are grouped by the type of lex-objective, where R indicates reachability, S safety. For each combination of case study and scaling parameters, we report the state size in column $|S|$, three different model checking runtimes, the average number of actions in the original and all considered restricted games, and the fraction of stages considered, i.e. the stages solved by the algorithm compared to the theoretically maximal possible number of stages $(2^n - 1)$.

We compare the time of our algorithm on the lexicographic objective (Lex.) to the time for checking the first single objective (First) and the sum of checking all single objectives (All). We see that the runtimes of our algorithm and checking all single objectives are always in the same order of magnitude. This shows that our algorithm works well in practice and that the overhead is often small. Even on SGs of non-trivial size (HW[10 × 10] and AV[20 × 20]), our algorithm returns the result within a few minutes.

Regarding the average number of actions, we see that the decrease in the number of actions in the sub-games $\widetilde{\mathcal{G}}$ obtained by restricting the input game to optimal actions varies: For example, very few actions are removed in the Dice instances, in AV we have a moderate decrease and in HW a significant decrease, almost eliminating all non-determinism after the first objective. It is our intuition that the less actions are removed, the higher is the overhead compared to the individual single-objective solutions. Consider the AV and HW examples: While for AV$[20 \times 20]$, computing the lexicographic solution takes 1.7 times as long as all the single-objective solutions, it took only about 25% longer for HW$[10 \times 10]$; this could be because in HW, after the first objective only little nondeterminism remains, while in AV also for the second and third objectives lots of choices have to be considered. Note that the first objective sometimes (HW), but not always (AV) needs the majority of the runtime.

We also see that the algorithm does not have to explore all possible stages. For example, for Dice we always just need a single stage, because the SG is absorbing. For charlton and HW all stages are relevant for the lex-objective, while for AV 4 of 7 need to be considered.

6 Conclusion and Future Work

In this work we considered simple stochastic games with lexicographic reachability and safety objectives. Simple stochastic games are a standard model in reactive synthesis of stochastic systems, and lexicographic objectives let one consider multiple objectives with an order of preference. We focused on the most basic objectives: safety and reachability. While simple stochastic games with lexicographic objectives have not been studied before, we have presented (a) determinacy; (b) strategy complexity; (c) computational complexity; and (d) algorithms; for these games. Moreover, we showed how these games can model many different case studies and we present experimental results for them.

There are several directions for future work. First, for the general case closing the complexity gap ($\mathsf{NEXPTIME} \cap \mathsf{coNEXPTIME}$ upper bound and PSPACE lower bound) is an open question. Second, the study of lexicographic simple stochastic games with more general objectives, e.g., quantitative or parity objectives poses interesting questions. In particular, in the case of parity objectives, there are some indications that the problem is significantly harder: Consider the case of a reachability-safety lex-objective. If the lex-value is $(1, 1)$ then both objectives can be guaranteed almost surely. Since almost-sure safety is sure safety, our results imply that sure safety and almost-sure reachability can be achieved with constant memory. In contrast, for parity objectives the combination of sure and almost-sure requires infinite-memory (e.g, see [21, Appendix A.1]).

References

1. Altman, E.: Constrained Markov Decision Processes. CRC Presss, Boca Raton (1999)

2. Baier, C., Dubslaff, C., Klüppelholz, S.: Trade-off analysis meets probabilistic model checking. In: CSL-LICS, pp. 1:1–1:10 (2014)
3. Baier, C., et al.: Probabilistic model checking and non-standard multi-objective reasoning. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 1–16. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_1
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Basset, N., Kwiatkowska, M., Topcu, U., Wiltsche, C.: Strategy synthesis for stochastic games with multiple long-run objectives. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 256–271. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_22
6. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_14
7. Blume, L., Brandenburger, A., Dekel, E.: Lexicographic probabilities and choice under uncertainty. Econometrica J. Econ. Soc. **59**(1), 61–79 (1991)
8. Brázdil, T., Brozek, V., Chatterjee, K., Forejt, V., Kucera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. LMCS **10**(1) (2014). https://doi.org/10.2168/LMCS-10(1:13)2014
9. Brázdil, T., Chatterjee, K., Forejt, V., Kucera, A.: Trading performance for stability in Markov decision processes. In: LICS, pp. 331–340 (2013)
10. Brázdil, T., Chatterjee, K., Forejt, V., Kučera, A.: MultiGain: a controller synthesis tool for MDPs with multiple mean-payoff objectives. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 181–187. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_12
11. Bruyère, V., Filiot, E., Randour, M., Raskin, J.: Meet your expectations with guarantees: beyond worst-case synthesis in quantitative games. Inf. Comput. **254**, 259–295 (2017)
12. Bruyère, V., Hautem, Q., Raskin, J.: Parameterized complexity of games with monotonically ordered omega-regular objectives. CoRR abs/1707.05968 (2017)
13. Chatterjee, K.: Markov decision processes with multiple long-run average objectives. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 473–484. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77050-3_39
14. Chatterjee, K., Chmelík, M.: POMDPs under probabilistic semantics. Artif. Intell. **221**, 46–72 (2015). https://doi.org/10.1016/j.artint.2014.12.009
15. Chatterjee, K., Chmelik, M., Gupta, R., Kanodia, A.: Optimal cost almost-sure reachability in POMDPs. Artif. Intell. **234**, 26–48 (2016). https://doi.org/10.1016/j.artint.2016.01.007
16. Chatterjee, K., Doyen, L.: Perfect-information stochastic games with generalized mean-payoff objectives. In: LICS. pp. 247–256. ACM (2016)
17. Chatterjee, K., Forejt, V., Wojtczak, D.: Multi-objective discounted reward verification in graphs and MDPs. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 228–242. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_17
18. Chatterjee, K., Henzinger, T.A.: A survey of stochastic ω-regular games. J. Comput. Syst. Sci. **78**(2), 394–413 (2012)
19. Chatterjee, K., Katoen, J.P., Weininger, M., Winkler, T.: Stochastic games with lexicographic reachability-safety objectives. CoRR abs/2005.04018 (2020). http://arxiv.org/abs/2005.04018

20. Chatterjee, K., Kretínská, Z., Kretínský, J.: Unifying two views on multiple mean-payoff objectives in Markov decision processes. LMCS **13**(2) (2017). https://doi.org/10.23638/LMCS-13(2:15)2017
21. Chatterjee, K., Piterman, N.: Combinations of qualitative winning for stochastic parity games. CoRR abs/1804.03453 (2018). http://arxiv.org/abs/1804.03453
22. Chatterjee, K., Sen, K., Henzinger, T.A.: Model-checking ω-regular properties of interval Markov chains. In: Amadio, R. (ed.) FoSSaCS 2008. LNCS, vol. 4962, pp. 302–317. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78499-9_22
23. Chen, T., Forejt, V., Kwiatkowska, M., Simaitis, A., Wiltsche, C.: On stochastic games with multiple objectives. In: Chatterjee, K., Sgall, J. (eds.) MFCS 2013. LNCS, vol. 8087, pp. 266–277. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40313-2_25
24. Chen, T., Kwiatkowska, M., Simaitis, A., Wiltsche, C.: Synthesis for multi-objective stochastic games: an application to autonomous urban driving. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 322–337. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_28
25. Colcombet, T., Jurdzinski, M., Lazic, R., Schmitz, S.: Perfect half space games. In: Logic in Computer Science, LICS 2017, pp. 1–11 (2017)
26. Condon, A.: The complexity of stochastic games. Inf. Comput. **96**(2), 203–224 (1992). https://doi.org/10.1016/0890-5401(92)90048-K
27. 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, Part II. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
28. Delgrange, F., Katoen, J.-P., Quatmann, T., Randour, M.: Simple strategies in multi-objective MDPs. In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12078, pp. 346–364. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_19
29. Etessami, K., Kwiatkowska, M.Z., Vardi, M.Y., Yannakakis, M.: Multi-objective model checking of Markov decision processes. LMCS **4**(4) (2008). https://doi.org/10.2168/LMCS-4(4:8)2008
30. Fijalkow, N., Horn, F.: The surprizing complexity of generalized reachability games. arXiv:1010.2420 [cs], October 2010
31. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer, New York (1997). https://doi.org/10.1007/978-1-4612-4054-9
32. Filar, J., Krass, D., Ross, K.: Percentile performance criteria for limiting average Markov decision processes. IEEE Trans. Autom. Control. **40**(1), 2–10 (1995)
33. Fishburn, P.C.: Exceptional paper – lexicographic orders, utilities and decision rules: a survey. Manag. Sci. **20**(11), 1442–1471 (1974)
34. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., Qu, H.: Quantitative multi-objective verification for probabilistic systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 112–127. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_11
35. Hartmanns, A., Junges, S., Katoen, J.-P., Quatmann, T.: Multi-cost bounded reachability in MDP. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, Part II. LNCS, vol. 10806, pp. 320–339. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_19
36. Hoffman, A.J., Karp, R.M.: On nonterminating stochastic games. Manag. Sci. **12**(5), 359–370 (1966). https://doi.org/10.1287/mnsc.12.5.359

37. Kwiatkowska, M., Parker, D., Wiltsche, C.: PRISM-games: verification and strategy synthesis for stochastic multi-player games with multiple objectives. STTT **20**(2), 195–210 (2018). https://doi.org/10.1007/s10009-017-0476-z
38. 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)
39. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (2014)
40. Quatmann, T., Junges, S., Katoen, J.-P.: Markov automata with multiple objectives. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017, Part I. LNCS, vol. 10426, pp. 140–159. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_7
41. Randour, M., Raskin, J.-F., Sankur, O.: Percentile queries in multi-dimensional Markov decision processes. Form. Methods Syst. Des. **50**(2–3), 207–248 (2017). https://doi.org/10.1007/s10703-016-0262-7
42. Roijers, D.M., Whiteson, S.: Multi-objective decision making. Synth. Lect. Artif. Intell. Mach. Learn. **11**(1), 1–129 (2017)
43. Svorenová, M., Kwiatkowska, M.: Quantitative verification and strategy synthesis for stochastic games. Eur. J. Control **30**, 15–30 (2016). https://doi.org/10.1016/j.ejcon.2016.04.009
44. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2), 285–309 (1955). https://doi.org/10.2140/pjm.1955.5.285
45. Weininger, M., Meggendorfer, T., Křetínský, J.: Satisfiability bounds for ω-regular properties in bounded-parameter Markov decision processes. In: CDC (2019, to appear)
46. Wray, K.H., Zilberstein, S.: Multi-objective POMDPs with lexicographic reward preferences. In: IJCAI, pp. 1719–1725. AAAI Press (2015)
47. Wray, K.H., Zilberstein, S., Mouaddib, A.: Multi-objective MDPs with conditional lexicographic reward preferences. In: AAAI, pp. 3418–3424. AAAI Press (2015)

Qualitative Controller Synthesis for Consumption Markov Decision Processes

František Blahoudek[1], Tomáš Brázdil[2], Petr Novotný[2], Melkior Ornik[3], Pranay Thangeda[3(✉)], and Ufuk Topcu[1]

[1] The University of Texas at Austin, Austin, USA
frantisek.blahoudek@gmail.com, utopcu@utexas.edu
[2] Masaryk University, Brno, Czech Republic
{xbrazdil,petr.novotny}@fi.muni.cz
[3] University of Illinois at Urbana-Champaign, Urbana, USA
{mornik,pranayt2}@illinois.edu

Abstract. Consumption Markov Decision Processes (CMDPs) are probabilistic decision-making models of resource-constrained systems. In a CMDP, the controller possesses a certain amount of a critical resource, such as electric power. Each action of the controller can consume some amount of the resource. Resource replenishment is only possible in special *reload states,* in which the resource level can be reloaded up to the full capacity of the system. The task of the controller is to prevent resource exhaustion, i.e. ensure that the available amount of the resource stays non-negative, while ensuring an additional linear-time property. We study the complexity of strategy synthesis in consumption MDPs with almost-sure Büchi objectives. We show that the problem can be solved in polynomial time. We implement our algorithm and show that it can efficiently solve CMDPs modelling real-world scenarios.

1 Introduction

In the context of formal methods, controller synthesis typically boils down to computing a strategy in an *agent-environment* model, a nondeterministic state-transition model where some of the nondeterministic choices are resolved by the controller and some by an uncontrollable environment. Such models are typically either two-player graph games with an adversarial environment or Markov decision process (MDPs); the latter case being apt for modelling statistically predictable environments. In this paper, we consider controller synthesis for *resource-constrained MDPs*, where the computed controller must ensure, in addition to satisfying some linear-time property, that the system's operation is not compromised by a lack of necessary resources.

This work was partially supported by NASA under Early Stage Innovations grant No. 80NSSC19K0209, and by DARPA under grant No. HR001120C0065. Petr Novotný is supported by the Czech Science Foundation grant No. GJ19-15134Y.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 421–447, 2020.
https://doi.org/10.1007/978-3-030-53291-8_22

Resource-Constrained Probabilistic Systems. Resource-constrained systems need a supply of some resource (e.g. power) for steady operation: the interruption of the supply can lead to undesirable consequences and has to be avoided. For instance, an autonomous system, e.g. an autonomous electric vehicle (AEV), is not able to draw power directly from an endless source. Instead, it has to rely on an internal storage of the resource, e.g. a battery, which has to be replenished in regular intervals to prevent resource exhaustion. Practical examples of AEVs include driverless cars, drones, or planetary rovers [8]. In these domains, resource failures may cause a costly mission failure and even safety risks. Moreover, the operation of autonomous systems is subject to probabilistic uncertainty [54]. Hence, in this paper, we study the resource-constrained strategy synthesis problem for MDPs.

Models of Resource-Constrained Systems & Limitations of Current Approaches. There is a substantial body of work in the area of verification of resource-constrained systems [3,5,7,9,11,23,38,39,53,58]. The typical approach is to model them as finite-state systems augmented with an integer-valued counter representing the current *resource level,* i.e. the amount of the resource present in the internal storage. The resource constraint requires that the resource level never drops below zero.[1] In the well-known *energy* model [11,23], each transition is labelled by an integer, and performing an ℓ-labelled transition results in ℓ being added to the counter. Thus, negative numbers stand for resource consumption while positive ones represent re-charging by the respective amount. Many variants of both MDP and game-based energy models were studied, as detailed in the related work. In particular, [26] considers controller synthesis for energy MDPs with qualitative Büchi and parity objectives. The main limitation of energy-based agent-environment models is that in general, they are not known to admit polynomial-time controller synthesis algorithms. Indeed, already the simplest problem, deciding whether a non-negative energy can be maintained in a two-player energy game, is at least as hard as solving mean-payoff graph games [11]; the complexity of the latter being a well-known open problem [45]. This hardness translates also to MDPs [26], making polynomial-time controller synthesis for energy MDPs impossible without a theoretical breakthrough.

Consumption models, introduced in [14], offer an alternative to energy models. In a consumption model, a non-negative integer, *cap*, represents the maximal amount of the resource the system can hold, e.g. the battery capacity. Each transition is labelled by a non-negative number representing the amount of the resource *consumed* when taking the transition (i.e., taking an ℓ-labelled transition decreases the resource level by ℓ). The resource replenishment is different from the energy approach. The consumption approach relies on the fact that reloads are often *atomic events*, e.g. an AEV plugging into a charging station and waiting to finish the charging cycle. Hence, some states in the consumption model are designated as *reload states,* and whenever the system visits a

[1] In some literature, the level is required to stay positive as opposed to non-negative, but this is only a matter of definition: both approaches are equivalent.

reload state, the resource level is replenished to the full capacity *cap*. Modelling reloads as atomic events is natural and even advantageous: consumption models typically admit more efficient analysis than energy models [14,47]. However, consumption models have not yet been considered in the probabilistic setting.

Our Contribution. We study strategy synthesis in consumption MDPs with Büchi objectives. Our main theoretical result is stated in the following theorem.

Theorem 1. *Given a consumption MDP $\mathcal{M}$ with a capacity cap, an initial resource level $0 \leq d \leq cap$, and a set T of accepting states, we can decide, in polynomial time, whether there exists a strategy σ such that when playing according to σ, the following* consumption-Büchi objectives *are satisfied:*

- *Starting with resource level d, the resource level never*[2] *drops below* 0.
- *With probability* 1*, the system visits some state in T infinitely often.*

Moreover, if such a strategy exists then we can compute, in polynomial time, its polynomial-size representation.

For the sake of clarity, we restrict to proving Theorem 1 for a natural sub-class of MDPs called *decreasing consumption MDPs,* where there are no cycles of zero consumption. The restriction is natural (since in typical resource-constrained systems, each action – even idling – consumes some energy, so zero cycles are unlikely) and greatly simplifies presentation. In addition to the theoretical analysis, we implemented the algorithm behind Theorem 1 and evaluated it on several benchmarks, including a realistic model of an AEV navigating the streets of Manhattan. The experiments show that our algorithm is able to efficiently solve large CMDPs, offering a good scalability.

Significance. Some comments on Theorem 1 are in order. First, all the numbers in the MDP, and in particular the capacity *cap*, are encoded in binary. Hence, "polynomial time" means time polynomial in the encoding size of the MDP itself and in $\log(cap)$. In particular, a naive "unfolding" of the MDP, i.e. encoding the resource levels between 0 and *cap* into the states, does not yield a polynomial-time algorithm, but an exponential-time one, since the unfolded MDP has size proportional to *cap*. We employ a value-iteration-like algorithm to compute minimal energy levels with which one can achieve the consumption-Büchi objectives.

A similar concern applies to the "polynomial-size representation" of the strategy σ. To satisfy a consumption-Büchi objective, σ generally needs to keep track of the current resource level. Hence, under the standard notion of a finite-memory (FM) strategy (which views FM strategies as transducers), σ would require memory proportional to *cap*, i.e. a memory exponentially large w.r.t. size of the input. However, we show that for each state s we can partition the integer interval $[0, \ldots, cap]$ into polynomially many sub-intervals $I_1^s, \ldots, I_k^s$ such that, for each $1 \leq j \leq k$, the strategy σ picks the same action whenever the current state is

[2] In our model, this is equivalent to requiring that with probability 1, the resource level never drops below 0.

s and the current resource level is in I_j^s. As such, the endpoints of the intervals are the only extra knowledge required to represent σ, a representation which we call a *counter selector*. We instrument our main algorithm so as to compute, in polynomial time, a polynomial-size counter selector representing the witness strategy σ.

Finally, we consider linear-time properties encoded by Büchi objectives over the states of the MDP. In essence, we assume that the translation of the specification to the Büchi automaton and its product with the original MDP model of the system were already performed. Probabilistic analysis typically requires the use of deterministic Büchi automata, which cannot express all linear-time properties. However, in this paper we consider qualitative analysis, which can be performed using restricted versions of non-deterministic Büchi automata that are still powerful enough to express all ω-regular languages. Examples of such automata are limit-deterministic Büchi automata [51] or good-for-MDPs automata [41]. Alternatively, consumption MDPs with parity objectives could be reduced to consumption-Büchi MPDs using the standard parity-to-Büchi MDP construction [25,30,32,33]. We abstract from these aspects and focus on the technical core of our problem, solving consumption-Büchi MDPs.

Consequently, to our best knowledge, we present the first polynomial-time algorithm for controller synthesis in resource-constrained MDPs with ω-regular objectives.

Related Work. There is an enormous body of work on energy models. Stemming from the models introduced in [11,23], the subsequent work covered energy games with various combinations of objectives [10,12,13,18,20,21,27,48], energy games with multiple resource types [15,24,28,31,37,43,44,57] or the variants of the above in the MDP [17,49], infinite-state [1], or partially observable [34] settings. As argued previously, the controller synthesis within these models is at least as hard as solving mean-payoff games. The paper [29] presents polynomial-time algorithms for non-stochastic energy games with special weight structures. Recently, an abstract algebraic perspective on energy models was presented in [22,35,36].

Consumption systems were introduced in [14] in the form of consumption games with multiple resource types. Minimizing mean-payoff in automata with consumption constraints was studied in [16].

Our main result requires, as a technical sub-component, solving the *resource-safety* (or just *safety*) problem in consumption MDPs, i.e. computing a strategy which prevents resource exhaustion. The solution to this problem consists (in principle) of a Turing reduction to the problem of minimum cost reachability in two-player games with non-negative costs. The latter problem was studied in [46], with an extension to arbitrary costs considered in [19] (see also [40]). We present our own, conceptually simple, value-iteration-like algorithm for the problem, which is also used in our implementation.

Elements of resource-constrained optimization and minimum-cost reachability are also present in the line of work concerning *energy-utility quantiles* in MDPs [4–7,42]. In this setting, there is no reloading in the consumption- or

energy-model sense, and the task is typically to minimize the total amount of the resource consumed while maximizing the probability that some other objective is satisfied.

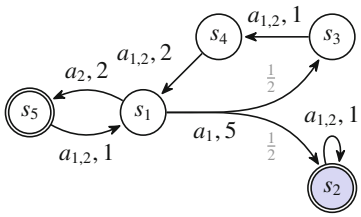
Paper Organization & Outline of Techniques. After the preliminaries (Sect. 2), we present counter selectors in Sect. 3. The next three sections contain the three main steps of our analysis. In Sect. 4, we solve the safety problem in consumption MDPs. The technical core of our approach is presented in Sect. 5, where we solve the problem of *safe positive reachability*: finding a resource-safe strategy which ensures that the set T of accepting states is visited with positive probability. Solving consumption-Büchi MDPs then, in principle, consists of repeatedly applying a strategy for safe positive reachability of T, ensuring that the strategy is "re-started" whenever the attempt to reach T fails. Details are given in Sect. 6. Finally, Sect. 7 presents our experiments. Due to space constraints, most technical proofs were moved to the full version.

2 Preliminaries

We denote by $\mathbb{N}$ the set of all non-negative integers and by $\overline{\mathbb{N}}$ the set $\mathbb{N} \cup \{\infty\}$. Given a set I and a vector $\mathbf{v} \in \overline{\mathbb{N}}^I$ of integers indexed by I, we use $\mathbf{v}(i)$ to denote the i-component of $\mathbf{v}$. We assume familiarity with basic notions of probability theory. In particular, a *probability distribution* on an at most countable set X is a function $f\colon X \to [0,1]$ s.t. $\sum_{x \in X} f(x) = 1$. We use $\mathcal{D}(X)$ to denote the set of all probability distributions on X.

Definition 1 (CMDP). *A* consumption Markov decision process *(CMDP) is a tuple* $\mathcal{M} = (S, A, \Delta, C, R, cap)$ *where* S *is a finite set of* states*,* A *is a finite set of* actions*,* $\Delta\colon S \times A \to \mathcal{D}(S)$ *is a total* transition function*,* $C\colon S \times A \to \mathbb{N}$ *is a total* consumption function*,* $R \subseteq S$ *is a set of* reload states *where the resource can be reloaded, and* cap *is a* resource capacity*.*

Figure 1 shows a visual representation of an CMDP. We denote by $\mathcal{M}(R')$ for $R' \subseteq S$ the CMDP obtained from $\mathcal{M}$ by changing the set of reloads to R'. For



Distributions in Δ are indicated by gray numbers (we leave out 1 when an action has only one successor), and the cost of an action follows its name in the edge labels. Actions labeled by $a_{1,2}$ represent that Δ and C are defined identically for both actions a_1 and a_2. The blue background indicates a target set $T = \{s_2\}$, while the double circles represent the reload states.

Fig. 1. CMDP $\mathcal{M} = (\{s_1, s_2, s_3, s_4, s_5\}, \{a_1, a_2\}, \Delta, C, \{s_2, s_5\}, 20)$. Details are given on the right.

$s \in S$ and $a \in A$, we denote by $Succ(s,a)$ the set $\{t \mid \Delta(s,a)(t) > 0\}$. A *path* is a (finite or infinite) state-action sequence $\alpha = s_1 a_1 s_2 a_2 s_3 \cdots \in (S \times A)^\omega \cup (S \cdot A)^* \cdot S$ such that $s_{i+1} \in Succ(s_i, a_i)$ for all i. We define $\alpha_i = s_i$ and $Act^i(\alpha) = a_i$. We use $\alpha_{..i}$ for the finite prefix $s_1 a_1 \dots s_i$ of α, $\alpha_{i..}$ for the suffix $s_i a_i \dots$, and $\alpha_{i..j}$ for the infix $s_i a_i \dots s_j$. A finite path is a *cycle* if it starts and ends in the same state and is *simple* if none of its infixes forms a cycle. The *length* of a path α is the number $len(\alpha)$ of actions on α and $len(\alpha) = \infty$ if α is infinite.

A CMDP is *decreasing* if for every cycle $s_1 a_1 s_2 \dots a_{k-1} s_k$ there exists $1 \leq i < k$ such that $C(s_i, a_i) > 0$. Throughout this paper we consider only decreasing CMDPs. The only place where this assumption is used are the proofs of Theorem 4 and Theorem 8.

An infinite path is called a *run*. We typically name runs by variants of the symbol ϱ. The set of all runs in $\mathcal{M}$ is denoted $\mathsf{Runs}_\mathcal{M}$. A finite path is called *history*. The set of all possible histories of $\mathcal{M}$ is $hist_\mathcal{M}$ or simply $hist$. We use $last(\alpha)$ for the last state of α. Let α be a history with $last(\alpha) = s_1$ and $\beta = s_1 a_1 s_2 a_2 \dots$; we define a *joint path* as $\alpha \odot \beta = \alpha a_1 s_2 a_2 \dots$.

A *strategy* for $\mathcal{M}$ is a function $\sigma \colon hist_\mathcal{M} \to A$ assigning to each history an action to play. A strategy is *memoryless* if $\sigma(\alpha) = \sigma(\beta)$ whenever $last(\alpha) = last(\beta)$. We do not consider randomized strategies in this paper, as they are non-necessary for qualitative ω-regular objectives on finite MDPs [30,32,33].

A computation of $\mathcal{M}$ under the control of a given strategy σ from some initial state $s \in S$ creates a path. The path starts with $s_1 = s$. Assume that the current path is α and let $s_i = last(\alpha)$ (we say that $\mathcal{M}$ is currently in s_i). Then the next action on the path is $a_i = \sigma(\alpha)$ and the next state s_{i+1} is chosen randomly according to $\Delta(s_i, a_i)$. Repeating this process *ad infinitum* yields an infinite sample run ϱ. We say that ϱ is *σ-compatible* if it can be produced using this process, and *s-initiated* if it starts in s. We denote the set of all σ-compatible, s-initiated runs by $\mathsf{Comp}_\mathcal{M}(\sigma, s)$.

We denote by $\mathbb{P}^\sigma_{\mathcal{M},s}(\mathsf{A})$ the probability that a sample run from $\mathsf{Comp}_\mathcal{M}(\sigma, s)$ belongs to a given measurable set of runs A. For details on the formal construction of measurable sets of runs as well as the probability measure $\mathbb{P}^\sigma_{\mathcal{M},s}$ see [2]. Throughout the paper, we drop the $_\mathcal{M}$ subscripts in symbols whenever $\mathcal{M}$ is known from the context.

2.1 Resource: Consumption, Levels, and Objectives

We denote by $cap(\mathcal{M})$ the battery capacity in the MDP $\mathcal{M}$. A resource is consumed along paths and can be reloaded in the reload states up to the full capacity. For a path $\alpha = s_1 a_1 s_2 \dots$ we define the consumption of α as $cons(\alpha) = \sum_{i=1}^{len(\alpha)} C(s_i, a_i)$ (since the consumption is non-negative, the sum is always well defined, though possibly diverging). Note that *cons* does not consider reload states at all. To accurately track the remaining amount of the resource, we use the concept of a *resource level*.

Definition 2 (Resource level). *Let $\mathcal{M}$ be a CMDP with a set of reload states R, let α be a history, and let $0 \leq d \leq cap(\mathcal{M})$ be an integer called* initial load.

Then the energy level after α initialized by d, *denoted by* $RL_d^{\mathcal{M}}(\alpha)$ *or simply as* $RL_d(\alpha)$, *is defined inductively as follows: for a zero-length history* s *we have* $RL_d^{\mathcal{M}}(s) = d$. *For a non-zero-length history* $\alpha = \beta a t$ *we denote* $c = C(last(\beta), a)$, *and put*

$$RL_d^{\mathcal{M}}(\alpha) = \begin{cases} RL_d^{\mathcal{M}}(\beta) - c & \text{if } last(\beta) \notin R \text{ and } c \leq RL_d^{\mathcal{M}}(\beta) \neq \bot \\ cap(\mathcal{M}) - c & \text{if } last(\beta) \in R \text{ and } c \leq cap(\mathcal{M}) \text{ and } RL_d^{\mathcal{M}}(\beta) \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Consider $\mathcal{M}$ from Fig. 1 and the history $\alpha(i) = (s_1 a_2 s_5 a_2)^i s_1$ with i as a parameter. We have $cons(\alpha(i)) = 3i$ and at the same time, following the inductive definition of $RL_d(\alpha(i))$ we have $RL_2(\alpha(i)) = 19$ for all $i \geq 1$ as the resource is reloaded every time in s_5. This generalizes into the following. Let α be a history and let $f, l \geq 0$ be the minimal and maximal indices i such that $\alpha_i \in R$, respectively. For $RL_d(\alpha) \neq \bot$, it holds $RL_d(\alpha_{..i}) = d - cons(\alpha_{..i})$ for all $i \leq f$ and $RL_d(\alpha) = cap(\mathcal{M}) - cons(\alpha_{l..})$. Further, for each history α and d such that $e = RL_d(\alpha) \neq \bot$, and each history β suitable for joining with α it holds that $RL_d(\alpha \odot \beta) = RL_e(\beta)$.

A run ϱ is *d-safe* if and only if the energy level initialized by d is a non-negative number for each finite prefix of ρ, i.e. if for all $i > 0$ we have $RL_d(\varrho_{..i}) \neq \bot$. We say that a run is safe if it is $cap(\mathcal{M})$-safe. The next lemma follows immediately from the definition of an energy level.

Lemma 1. *Let* $\varrho = s_1 a_1 s_2 \ldots$ *be a d-safe run for some* d *and let* α *be a history such that* $last(\alpha) = s_1$. *Then the run* $\alpha \odot \varrho$ *is e-safe if* $RL_e(\alpha) \geq d$.

Example 1. Recall the CMDP and the parameterized history $\alpha(i)$ from above. We know that $RL_2(\alpha(i)) = 19$ for all i. Therefore, a strategy that always picks a_2 in s_1 is d-safe in s_1 for all $d \geq 2$. On the other hand, a strategy that always picks a_1 in s_1 is *not* d-safe in s_1 for any $0 \leq d \leq 20 = cap(\mathcal{M})$ because for all runs ϱ that visit s_3 at least three times before s_2 we have $RL_d(\varrho) = \bot$.

Objectives. An *objective* is a set of runs. The objective $\mathsf{SafeRuns}(d)$ contains exactly d-safe runs. Given a *target set* $T \subseteq S$ and $i \in \mathbb{N}$, we define $\mathsf{Reach}_T^i = \{\varrho \in \mathsf{Runs} \mid \varrho_j \in T \text{ for some } 1 \leq j \leq i+1\}$ to be the set of all runs that reach some state from T within the first i steps. We put $\mathsf{Reach}_T = \bigcup_{i \in \mathbb{N}} \mathsf{Reach}_T^i$. Finally, the set $\mathsf{Büchi}_T = \{\varrho \in \mathsf{Runs} \mid \varrho_i \in T \text{ for infinitely many } i \in \mathbb{N}\}$.

Problems. We solve three main qualitative problems for CMDPs, namely *safety*, *positive reachability*, and *Büchi*.

Let us fix a state s and a target set of states T. We say that a strategy σ is *d-safe in* s if $\mathsf{Comp}(\sigma, s) \subseteq \mathsf{SafeRuns}(d)$. We say that σ is *T-positive d-safe in* s if it is d-safe in s and $\mathbb{P}_s^\sigma(\mathsf{Reach}_T) > 0$, which means that there exists a run in $\mathsf{Comp}(\sigma, s)$ that visits T. Finally, we say that σ is *T-Büchi d-safe in a state* s if it is d-safe in s and $\mathbb{P}_s^\sigma(\mathsf{Büchi}_T) = 1$.

The vectors *Safe*, $SafePR_T$ (PR for "positive reachability"), and $SafeB\ddot{u}chi_T$ of type $\overline{\mathbb{N}}^S$ contain, for each $s \in S$, the minimal d such that there exists a strategy

that is d-safe in s, T-positive d-safe in s, and T-Büchi d-safe in s, respectively, and ∞ if no such strategy exists.

The problems we consider for a given CMDP are:

- *Safety:* compute the vector *Safe* and a strategy that is $Safe(s)$-safe in every $s \in S$.
- *Positive reachability:* compute the vector $SafePR_T$ and a strategy that is T-positive $SafePR_T(s)$-safe in every state s.
- *Büchi:* compute $SafeB\ddot{u}chi_T$ and a strategy that is T-Büchi $SafeB\ddot{u}chi_T(s)$-safe in every state s.

Example 2. Now consider again the d-safe strategy from Example 1 that always picks a_2; such a strategy is 2-safe in s_1, but is not useful if we attempt to eventually reach T. Hence memoryless strategies are not sufficient in our setting. Consider, instead, a strategy σ that picks a_1 in s_1 whenever the current resource level is at least 10 and picks a_2 otherwise. Such a strategy is 2-safe in s_1 and guarantees reaching s_2 with a positive probability: we need at least 10 units of energy to return to s_5 in the case we are unlucky and picking a_1 leads us to s_3. If we are lucky, a_1 leads us to s_2 by consuming just 5 units of the resource, witnessing that σ is T-positive. As a matter of fact, during *every* revisit of s_5 there is a $\frac{1}{2}$ chance of hitting s_2 during the next try, so σ actually ensures that s_2 is visited with probability 1.

Solving a CMDP is substantially different from solving a consumption 2-player game [14]. Indeed, imagine that in $\mathcal{M}$ from Fig. 1, the outcome of the action a_1 from state s_1 is resolved by an adversarial player. In such a game, the strategy σ does not produce any run that reaches s_2. In fact, there would be no strategy that guarantees reaching T in a 2-player game like this at all.

The strategy σ from our example uses finite memory to track the resource level exactly. We describe an efficient representation of such strategies in the next section.

3 Counter Strategies

In this section, we define a succinct representation of finite-memory strategies via so called counter selectors. Under the standard definition, a strategy σ is a *finite memory* strategy, if σ can be encoded by a *memory structure*, a type of finite transducer. Formally, a memory structure is a tuple $\mu = (M, nxt, up, m_0)$ where M is a finite set of *memory elements*, $nxt\colon M \times S \to A$ is a *next action* function, $up\colon M \times S \times A \times S \to M$ is a *memory update* function, and $m_0\colon S \to M$ is the *memory initialization function*. The function up can be lifted to a function $up^*\colon M \times hist \to M$ as follows.

$$up^*(m, \alpha) = \begin{cases} m & \text{if } \alpha = s \text{ has length } 0 \\ up\big(up^*(m, \beta), last(\beta), a, t\big) & \text{if } \alpha = \beta a t \text{ for some } a \in A \text{ and } t \in S \end{cases}$$

The structure μ encodes a strategy σ_μ such that for each history $\alpha = s_1 a_1 s_2 \dots s_n$ we have $\sigma_\mu(\alpha) = nxt\big(up^*(m_0(s_1), \alpha), s_n\big)$.

In our setting, strategies need to track energy levels of histories. Let us fix an CMDP $\mathcal{M} = (S, A, \Delta, C, R, cap)$. A non-exhausted energy level is always a number between 0 and $cap(\mathcal{M})$, which can be represented with a binary-encoded bounded counter. We call strategies with such counters *finite counter (FC) strategies*. An FC strategy selects actions to play according to *selection rules*.

Definition 3 (Selection rule). *A* selection rule φ *for* $\mathcal{M}$ *is a partial function from the set* $\{0, \ldots, cap(\mathcal{M})\}$ *to* A*. Undefined value for some* n *is indicated by* $\varphi(n) = \bot$*.*

We use $dom(\varphi) = \{n \in \{0, \ldots, cap(\mathcal{M})\} \mid \varphi(n) \neq \bot\}$ to denote the domain of φ and we use $Rules_{\mathcal{M}}$ or simply $Rules$ for the set of all selection rules for $\mathcal{M}$. Intuitively, a selection according to rule φ selects the action that corresponds to the largest value from $dom(\varphi)$ that is not larger than the current energy level. To be more precise, if $dom(\varphi)$ consists of numbers $n_1 < n_2 < \cdots < n_k$, then the action to be selected in a given moment is $\varphi(n_i)$, where n_i is the largest element of $dom(\varphi)$ which is less then or equal to the current amount of the resource. In other words, $\varphi(n_i)$ is to be selected if the current resource level is in $[n_i, n_{i+1})$ (putting $n_{k+1} = \infty$).

Definition 4 (Counter selector). *A* counter selector *for* $\mathcal{M}$ *is a function* $\Sigma \colon S \to Rules$*.*

A counter selector itself is not enough to describe a strategy. A strategy needs to keep track of the energy level throughout the path. With a vector $\mathbf{r} \in \{0, \ldots, cap(\mathcal{M})\}^S$ of initial resource levels, each counter selector Σ defines a strategy $\Sigma^{\mathbf{r}}$ that is encoded by the following memory structure (M, nxt, up, m_0) with $a \in A$ being a globally fixed action (for uniqueness). We stipulate that $\bot < n$ for all $n \in \mathbb{N}$.

- $M = \{\bot\} \cup \{0, \ldots, cap(\mathcal{M})\}$.
- Let $m \in M$ be a memory element, let $s \in S$ be a state, let $n \in dom(\Sigma(s))$ be the largest element of $dom(\Sigma(s))$ such that $n \leq m$. Then $nxt(m, s) = \Sigma(s)(n)$ if n exists, and $nxt = a$ otherwise.
- The function up is defined for each $m \in M, a \in A, s, t \in S$ as follows.

$$up(m, s, a, t) = \begin{cases} m - C(s, a) & \text{if } s \notin R \text{ and } C(s, a) \leq m \neq \bot \\ cap(\mathcal{M}) - C(s, a) & \text{if } s \in R \text{ and } C(s, a) \leq cap(\mathcal{M}) \text{ and } m \neq \bot \\ \bot & \text{otherwise.} \end{cases}$$

- The function m_0 is $m_0(s) = \mathbf{r}(s)$.

A strategy σ is a finite counter (FC) strategy if there is a counter selector Σ and a vector $\mathbf{r}$ such that $\sigma = \Sigma^{\mathbf{r}}$. The counter selector can be imagined as a finite-state device that implements σ using $\mathcal{O}(\log(cap(\mathcal{M})))$ bits of additional memory (counter) used to represent numbers $0, 1, \ldots, cap(\mathcal{M})$. The device uses the counter to keep track of the current resource level, the element $\bot$ representing energy exhaustion. Note that a counter selector can be exponentially more succinct than the corresponding memory structure.

Example 3. Consider again the CMDP $\mathcal{M}$ in Fig. 1 and a counter selector Σ defined as follows: Let φ be a selection rule with $dom(\varphi) = \{0, 10\}$ such that $\varphi(0) = a_2$ and $\varphi(10) = a_1$. Then let φ' be a selection rule such that $dom(\varphi') = \{0\}$ and $\varphi(0) = a_1$. Finally, let Σ be a counter selector such that $\Sigma(s_1) = \varphi$ and $\Sigma(s_i) = \varphi'$ for all $i \neq 1$. Then, for a vector of initial resource levels $\mathbf{r}$, the strategy σ informally described in Example 2 can be formally represented by putting $\sigma = \Sigma^{\mathbf{r}}$. Note that for any $\mathbf{r}$ with $\mathbf{r}(s_1) \geq 2$, $\mathbf{r}(s_2) \geq 0$, $\mathbf{r}(s_3) \geq 5$, $\mathbf{r}(s_4) \geq 4$, and $\mathbf{r}(s_5) \geq 0$ and for any state s of $\mathcal{M}$ the strategy $\Sigma^{\mathbf{r}}$ is $\mathbf{r}(s)$-safe in s.

4 Safety

In this section, we present an algorithm that computes, for each state, the minimal value d (if it exists) such that there exists a d-safe strategy from that state. We also provide the corresponding strategy. In the remainder of the section we fix an MDP $\mathcal{M}$.

A d-safe run has the following two properties: (i) It consumes at most d units of the resource (energy) before it reaches the first reload state, and (ii) it never consumes more than $cap(\mathcal{M})$ units of the resource between 2 visits of reload states. To ensure (ii), we need to identify a maximal subset $R' \subseteq R$ of reload states for which there is a strategy σ that, starting in some $r \in R'$, can always reach R' again (within at least one step) using at most $cap(\mathcal{M})$ resource units. The d-safe strategy we seek can be then assembled from σ and from a strategy that suitably navigates towards R', which is needed for (i).

In the core of both properties (i) and (ii) lies the problem of *minimum cost reachability.* Hence, in the next subsection, we start with presenting necessary results on this problem.

4.1 Minimum Cost Reachability

The problem of minimum cost reachability with non-negative costs was studied before [46]. Here we present a simple approach to the problem used in our implementation and most of the technical details are available in the full version.

Definition 5. *Let $T \subseteq S$ be a set of* target *states, let $\alpha = s_1 a_1 s_2 \ldots$ be a finite or infinite path, and let $1 \leq f$ be the smallest index such that $s_f \in T$. We define* consumption of α to T *as $ReachCons_{\mathcal{M},T}(\alpha) = cons(\alpha_{..f})$ if f exists and we set $ReachCons_{\mathcal{M},T}(\alpha) = \infty$ otherwise. For a strategy σ and a state $s \in S$ we define $ReachCons_{\mathcal{M},T}(\sigma, s) = \sup_{\varrho \in \mathsf{Comp}(\sigma,s)} ReachCons_{\mathcal{M},T}(\varrho)$.*
A minimum cost reachability of T from s *is a vector defined as*

$$MinReach_{\mathcal{M},T}(s) = \inf \left\{ ReachCons_{\mathcal{M},T}(\sigma, s) \mid \sigma \text{ is a strategy for } \mathcal{M} \right\}.$$

Intuitively, $d = MinReach_T(s)$ is the minimal initial load with which some strategy can ensure reaching T with consumption at most d, when starting

in s. We say that a strategy σ is optimal for $MinReach_T$ if we have that $MinReach_T(s) = ReachCons_T(\sigma, s)$ for all states $s \in S$.

We also define functions $ReachCons^+_{\mathcal{M},T}$ and the vector $MinReach^+_{\mathcal{M},T}$ in a similar fashion with one exception: we require the index f from definition of $ReachCons_{\mathcal{M},T}(\alpha)$ to be strictly larger than 1, which enforces to take at least one step to reach T.

For the rest of this section, fix a target set T and consider the following functional $\mathcal{F}$:

$$\mathcal{F}(\mathbf{v})(s) = \begin{cases} \min_{a \in A} \left(C(s,a) + \max_{t \in Succ(s,a)} \mathbf{v}(t) \right) & s \notin T \\ 0 & s \in T \end{cases}$$

$\mathcal{F}$ is a simple generalization of the standard Bellman functional used for computing shortest paths in graphs. The proof of the following Theorem is rather standard and moved to the full version of the paper.

Theorem 2. *Denote by n the length of the longest simple path in $\mathcal{M}$. Let $\mathbf{x}_T$ be a vector such that $\mathbf{x}_T(s) = 0$ if $s \in T$ and $\mathbf{x}_T(s) = \infty$ otherwise. Then iterating $\mathcal{F}$ on $\mathbf{x}_T$ yields a fixpoint in at most n steps and this fixpoint equals $MinReach_T$.*

To compute $MinReach^+_{\mathcal{M},T}$, we construct a new CMDP $\widetilde{\mathcal{M}}$ from $\mathcal{M}$ by adding a copy $\tilde{s}$ of each state $s \in S$ such that dynamics in $\tilde{s}$ is the same as in s; i.e. for each $a \in A$, $\Delta(\tilde{s}, a) = \Delta(s, a)$ and $C(\tilde{s}, a) = C(s, a)$. We denote the new state set as $\widetilde{S}$. We don't change the set of reload states, so $\tilde{s}$ is *never* in T, even if s is. Given the new CMDP $\widetilde{\mathcal{M}}$ and the new state set as $\widetilde{S}$, the following lemma is straightforward.

Lemma 2. *Let $\mathcal{M}$ be a CMDP and let $\widetilde{\mathcal{M}}$ be the CMDP constructed as above. Then for each state s of $\mathcal{M}$ it holds $MinReach^+_{\mathcal{M},T}(s) = MinReach_{\widetilde{\mathcal{M}},T}(\tilde{s})$.*

4.2 Safely Reaching Reload States

In the following, we use $MinInitCons_{\mathcal{M}}$ (read *minimal initial consumption*) for the vector $MinReach^+_{\mathcal{M},R}$ – minimal resource level that ensures we can surely reach a reload state in at least one step. By Lemma 2 and Theorem 2 we can construct $\widetilde{\mathcal{M}}$ and iterate the operator $\mathcal{F}$ for $|S|$ steps to compute $MinInitCons_{\mathcal{M}}$. Note that S is the state space of $\mathcal{M}$ since introducing the new states into $\widetilde{\mathcal{M}}$ did not increase the length of the maximal simple path. However, we can avoid the construction of $\widetilde{\mathcal{M}}$ and still compute $MinInitCons_{\mathcal{M}}$ using a *truncated* version of the functional $\mathcal{F}$, which is the approach used in our implementation. We first introduce the following truncation operator:

$$[\![\mathbf{x}]\!]_{\mathcal{M}}(s) = \begin{cases} \mathbf{x}(s) & \text{if } s \notin R, \\ 0 & \text{if } s \in R. \end{cases}$$

Algorithm 1: Algorithm for computing $MinInitCons_{\mathcal{M}}$.

Input: CMDP $\mathcal{M} = (S, A, \Delta, C, R, cap)$
Output: The vector $MinInitCons_{\mathcal{M}}$

1 initialize $\mathbf{x} \in \overline{\mathbb{N}}^S$ to be ∞ in every component;
2 **repeat**
3 $\mathbf{x}_{old} \leftarrow \mathbf{x}$;
4 **foreach** $s \in S$ **do**
5 $c \leftarrow \min_{a \in A} \left\{ C(s,a) + \max_{s' \in Succ(s,a)} [\![\mathbf{x}_{old}]\!]_{\mathcal{M}}(s') \right\}$;
6 **if** $c < \mathbf{x}(s)$ **then**
7 $\mathbf{x}(s) \leftarrow c$;
8 **until** $\mathbf{x}_{old} = \mathbf{x}$;
9 **return** $\mathbf{x}$

Then, we define a truncated functional $\mathcal{G}$ as follows:

$$\mathcal{G}(\mathbf{v})(s) = \min_{a \in A} \left(C(s,a) + \max_{s' \in Succ(s,a)} [\![\mathbf{v}]\!]_{\mathcal{M}}(s') \right).$$

The following lemma connects the iteration of $\mathcal{G}$ on $\mathcal{M}$ with the iteration of $\mathcal{F}$ on $\widetilde{\mathcal{M}}$.

Lemma 3. *Let $\boldsymbol{\infty} \in \overline{\mathbb{N}}^S$ be a vectors with all components equal to ∞. Consider iterating $\mathcal{G}$ on $\boldsymbol{\infty}$ in $\mathcal{M}$ and $\mathcal{F}$ on $\mathbf{x}_R$ in $\widetilde{\mathcal{M}}$. Then for each $i \geq 0$ and each $s \in R$ we have $\mathcal{G}^i(\boldsymbol{\infty})(s) = \mathcal{F}^i(\mathbf{x}_R)(\tilde{s})$ and for every $s \in S \setminus R$ we have $\mathcal{G}^i(\boldsymbol{\infty})(s) = \mathcal{F}^i(\mathbf{x}_R)(s)$.*

Algorithm 1 uses $\mathcal{G}$ to compute the vector $MinInitCons_{\mathcal{M}}$.

Theorem 3. *Algorithm 1 correctly computes the vector $MinInitCons_{\mathcal{M}}$. Moreover, the repeat-loop terminates after at most $|S|$ iterations.*

4.3 Solving the Safety Problem

We want to identify a set $R' \subseteq R$ such that we can reach R' in at least 1 step and with consumption at most $cap = cap(\mathcal{M})$, from each $r \in R'$. This entails identifying the maximal $R' \subseteq R$ such that $MinInitCons_{\mathcal{M}(R')} \leq cap$ for each $r \in R'$. This can be done by initially setting $R' = R$ and iteratively removing states that have $MinInitCons_{\mathcal{M}(R')} > cap$, from R', as in Algorithm 2.

Theorem 4. *Algorithm 2 computes the vector $Safe_{\mathcal{M}}$ in polynomial time.*

Proof. The algorithm clearly terminates. Computing $MinInitCons_{\mathcal{M}(Rel)}$ on line 5 takes a polynomial number of steps per call due to Theorem 3 and since $\mathcal{M}(Rel)$ has asymptotically the same size as $\mathcal{M}$. Since the repeat loop performs at most $|R|$ iterations, the complexity follows.

Algorithm 2: Computing the vector $Safe_{\mathcal{M}}$.

Input: CMDP $\mathcal{M}$
Output: The vector $Safe_{\mathcal{M}}$

```
1  cap ← cap(M);
2  Rel ← R; ToRemove ← ∅;
3  repeat
4  |  Rel ← Rel ∖ ToRemove;
5  |  mic ← MinInitCons_{M(Rel)};
6  |  ToRemove ← {r ∈ Rel | mic(r) > cap};
7  until ToRemove = ∅;
8  foreach s ∈ S do
9  |  if mic(s) > cap then out(s) = ∞;
10 |  else out(s) = mic(s);
11 return out
```

As for correctness, we first prove that $\mathbf{out} \leq Safe_{\mathcal{M}}$. It suffices to prove for each $s \in S$ that upon termination, $\mathbf{mic}(s) \leq Safe_{\mathcal{M}}(s)$ whenever the latter value is finite. Since $MinInitCons_{\mathcal{M}'}(s) \leq Safe_{\mathcal{M}'}(s)$ for each MDP $\mathcal{M}'$ and each its state such that $Safe_{\mathcal{M}'}(s) < \infty$, it suffices to show that $Safe_{\mathcal{M}(Rel)} \leq Safe_{\mathcal{M}}$ is an invariant of the algorithm (as a matter of fact, we prove that $Safe_{\mathcal{M}(Rel)} = Safe_{\mathcal{M}}$). To this end, it suffices to show that at every point of execution $Safe_{\mathcal{M}}(t) = \infty$ for each $t \in R \setminus Rel$: indeed, if this holds, no strategy that is safe for some state $s \neq t$ can play an action a from s such that $t \in Succ(s, a)$, so declaring such states non-reloading does not influence the $Safe_{\mathcal{M}}$-values. So denote by Rel_i the contents of Rel after the i-th iteration. We prove, by induction on i, that $Safe_{\mathcal{M}}(s) = \infty$ for all $s \in R \setminus Rel$. For $i = 0$ we have $R = Rel$, so the statement holds. For $i > 0$, let $s \in R \setminus Rel_i$, and let σ be any strategy. If some run from $\mathsf{Comp}(\sigma, s)$ visits a state from $R \setminus Rel_{i-1}$, then σ is not cap-safe, by induction hypothesis. Now assume that all such runs only visit reload states from Rel_{i-1}. Then, since $MinInitCons_{\mathcal{M}(Rel_{i-1})}(s) > cap$, there must be a run $\varrho \in \mathsf{Comp}(\sigma, s)$ with $ReachCons^{+}_{Rel_{i-1}}(\varrho) > cap$. Assume that ϱ is cap-safe in s. Since we consider only decreasing CMDPs, ϱ must infinitely often visit a reload state (as it cannot get stuck in a zero cycle). Hence, there exists an index $f > 1$ such that $\varrho_f \in Rel_{i-1}$, and for this f we have $RL_{cap}(\varrho_{..f}) = \bot$, a contradiction. So again, σ is not safe in s. Since there is no safe strategy from s, we have $Safe_{\mathcal{M}}(s) = \infty$.

Finally, we need to prove that upon termination, $\mathbf{out} \geq Safe_{\mathcal{M}}$. Informally, per the definition of **out**, from every state s we can ensure reaching a state of Rel by consuming at most $\mathbf{out}(s)$ units of the resource. Once in Rel, we can ensure that we can again return to Rel without consuming more than cap units of the resource. Hence, when starting with $\mathbf{out}(s)$ units, we can surely prevent resource exhaustion. □

Definition 6. *We call an action a* safe *in a state s if one of the following conditions holds:*

– $s \notin R$ *and* $C(s,a) + \max_{t \in Succ(s,a)} Safe_{\mathcal{M}}(t) \leq Safe_{\mathcal{M}}(s)$*; or*
– $s \in R$ *and* $C(s,a) + \max_{t \in Succ(s,a)} Safe_{\mathcal{M}}(t) \leq cap(\mathcal{M})$.

Note that by the definition of $Safe_{\mathcal{M}}$*, for each state* s *with* $Safe_{\mathcal{M}}(s) < \infty$ *there is always at least one action safe in* s*. For states* s *s.t.* $Safe_{\mathcal{M}}(s) = \infty$*, we stipulate all actions to be safe in* s*.*

Theorem 5. *Any strategy which always selects an action that is safe in the current state is* $Safe_{\mathcal{M}}(s)$*-safe in every state* s*. In particular, in each consumption MDP* $\mathcal{M}$ *there is a memoryless strategy* σ *that is* $Safe_{\mathcal{M}}(s)$*-safe in every state* s*. Moreover,* σ *can be computed in polynomial time.*

Proof. The first part of the theorem follows directly from Definition 6, Definition 2 (resource levels), and from definition of d-safe runs. The second part is a corollary of Theorem 4 and the fact that in each state, the safe strategy from Definition 6 can fix one such action in each state and thus is memoryless. The complexity follows from Theorem 4. □

Example 4. Consider again the $\mathcal{M}$ from Fig. 1. *Algorithm* 1 returns, for input $\mathcal{M}$, the vector $\mathbf{mic} = (2, 1, 5, 4, 3)$. Algorithm 2 reuses $\mathbf{mic}$ on line 5 and returns it unchanged. Hence, the vector $\mathbf{mic}$ equals $Safe_{\mathcal{M}}$. The strategies described in Example 1 witness that $Safe(s_1) \leq 2$. Here we see that there is no strategy that would be 1-safe in s_1.

5 Positive Reachability

In this section, we focus on strategies that are safe and such that at least one run they produce visits a given set $T \subseteq S$ of *targets*. The main contribution of this section is Algorithm 3 used to compute such strategies as well as the vector $SafePR_{\mathcal{M},T}$ of minimal initial resource levels for which such a strategy exist. As before, for the rest of this section we fix a CMDP $\mathcal{M}$.

We define a function $SPR\text{-}Val_{\mathcal{M}} \colon S \times A \times \overline{\mathbb{N}}^S \to \overline{\mathbb{N}}$ (*SPR* for safe positive reachability) s.t. for all $s \in S, a \in A$, and $\mathbf{x} \in \overline{\mathbb{N}}^S$ we have

$$SPR\text{-}Val_{\mathcal{M}}(s,a,\mathbf{x}) = C(s,a) + \min_{t \in Succ(s,a)} \left\{ \max\left\{ \mathbf{x}(t), Safe_{\mathcal{M}}(t') \mid t' \in Succ(s,a), t' \neq t \right\} \right\}$$

The max operator considers, for given t, the value $\mathbf{x}(t)$ and the values needed to survive from all possible outcomes of a other than t. Let $v = SPR\text{-}Val_{\mathcal{M}}(s,a,\mathbf{x})$ and t the outcome selected by min. Intuitively, v is the minimal amount of resource needed to reach t with at least $\mathbf{x}(t)$ resource units, or survive if the outcome of a is different from t.

We now define a functional whose fixed point characterizes $SPR\text{-}Val_{\mathcal{M},T}$. We first define a two-sided version of the truncation operator from the previous section: the operator $[\![\cdot]\!]_{\mathcal{M}}$ such that

$$[\![\mathbf{x}]\!]_{\mathcal{M}}(s) = \begin{cases} \infty & \text{if } \mathbf{x}(s) > cap(\mathcal{M}) \\ \mathbf{x}(s) & \text{if } \mathbf{x}(s) \leq cap(\mathcal{M}) \text{ and } s \notin R \\ 0 & \text{if } \mathbf{x}(s) \leq cap(\mathcal{M}) \text{ and } s \in R \end{cases}$$

Using the functions *SPR-Val* and $[\![\cdot]\!]_{\mathcal{M}}$, we now define an auxiliary operator $\mathcal{A}$ and the main operator $\mathcal{B}$ as follows.

$$\mathcal{A}_{\mathcal{M}}(\mathbf{r})(s) = \begin{cases} \mathit{Safe}_{\mathcal{M}}(s) & \text{if } s \in T \\ \min_{a \in A} (\mathit{SPR\text{-}Val}_{\mathcal{M}}(s, a, \mathbf{r})) & \text{otherwise;} \end{cases}$$
$$\mathcal{B}_{\mathcal{M}}(\mathbf{r}) = [\![\mathcal{A}_{\mathcal{M}}(\mathbf{r})]\!]_{\mathcal{M}}$$

Let SafePR^i_T be the vector such that for a state $s \in S$ the number $d = \mathit{SafePR}^i_T(s)$ is the minimal number ℓ such that there exists a strategy that is ℓ-safe in s and produces at least one run that visits T within first i steps. Further, we denote by $\mathbf{y}_T$ a vector such that

$$\mathbf{y}_T(s) = \begin{cases} \mathit{Safe}_{\mathcal{M}}(s) & \text{if } s \in T \\ \infty & \text{if } s \notin T \end{cases}$$

The following lemma can proved by a rather straightforward but technical induction.

Lemma 4. *Consider the iteration of $\mathcal{B}_{\mathcal{M}}$ on the initial vector $\mathbf{y}_T$. Then for each $i \geq 0$ it holds that $\mathcal{B}^i_{\mathcal{M}}(\mathbf{y}_T) = \mathit{SafePR}^i_{\mathcal{M},T}$.*

The following lemma says that iterating $\mathcal{B}_{\mathcal{M}}$ reaches a fixed point in a polynomial number of iterations. Intuitively, this is because when trying to reach T, it doesn't make sense to perform a cycle between two visits of a reload state (as this can only increase the resource consumption) and at the same time it doesn't make sense to visit the same reload state twice (since the resource is reloaded to the full capacity upon each visit). The proof is straightforward and is omitted in the interest of brevity. Detailed proofs for Lemma 4 and Lemma 5 are available in the full version of the paper.

Lemma 5. *Let $K = |R| + (|R| + 1) \cdot (|S| - |R| + 1)$. Taking the same initial vector $\mathbf{y}_T$ as in Lemma 4, we have $\mathcal{B}^K_{\mathcal{M}}(\mathbf{y}_T) = \mathit{SafePR}_{\mathcal{M},T}$.*

The computation of $\mathit{SafePR}_{\mathcal{M},T}$ and of the associated witness strategy is presented in Algorithm 3.

Example 5. Consider again the CMDP $\mathcal{M}$ from Fig. 1. After one iteration of the loop on line 5, we have $\mathbf{r} = (10, 0, \infty, \infty, \infty)$, as $\mathbf{r}$ is only finite for s_2 before this iteration. In the next iteration, we have $\mathbf{r} = (10, 0, \infty, 12, 0)$. Thus, the next iteration changes the value for s_1 to 2 and in the end, we end up with $\mathbf{r} = (2, 0, 4, 5, 0)$. The iteration with $\mathbf{r}(s_1) = 10$ influences the selector Σ. Note that the computed $\mathbf{r}$ and Σ match those mentioned in Example 3.

Theorem 6. *The Algorithm 3 always terminates after a polynomial number of steps, and upon termination, $\mathbf{r} = \mathit{SafePR}_{\mathcal{M},T}$.*

Algorithm 3: Positive reachability of T in $\mathcal{M}$

Input: CMDP $\mathcal{M}$ with states S, set of target states $T \subseteq S$
Output: The vector $SafePR_{\mathcal{M},T}$, coreresponding rule selector Σ

```
1  r ← {∞}^S;
2  foreach s ∈ S s.t. Safe_M(s) < ∞ do
3  |  Σ(s)(Safe_M(s)) ← arbitrary action safe in s
4  foreach t ∈ T do  r(t) ← Safe_M(t) ;
5  repeat
6  |  r_old ← r;
7  |  foreach s ∈ S \ T do
8  |  |  a(s) ← arg min_{a∈A} SPR-Val(s, a, r_old);
9  |  |  r(s) ← min_{a∈A} SPR-Val(s, a, r_old);
10 |  r ← [[ r ]]_M;
11 |  foreach s ∈ S \ T do
12 |  |  if r(s) < r_old(s)  then
13 |  |  |  Σ(s)(r(s)) ← a(s);
14 until r_old = r;
15 return r, Σ
```

Proof. The repeat loop on lines 1–4 initialize $\mathbf{r}$ to $\mathbf{y}_T$. The repeat loop on lines 5–14 then iterates the operator $\mathcal{B}$. By Lemma 5, the iteration reaches a fixed point in at most K steps, and this fixed point equals $SafePR_{\mathcal{M},T}$. The complexity bound follows easily, since K is of polynomial magnitude.

The most intricate part of our analysis is extracting a strategy that is T-positive $SafePR_{\mathcal{M},T}(s)$-safe in every state s.

Theorem 7. *Let $\mathbf{v} = SafePR_{\mathcal{M},T}$. Upon termination of Algorithm 3, the computed selector Σ has the property that the finite counter strategy $\Sigma^{\mathbf{v}}$ is, for each state $s \in S$, T-positive $\mathbf{v}(s)$-safe in s. That is, a polynomial-size finite counter strategy for the positive reachability problem can be computed in polynomial time.*

The rest of this section is devoted to the proof of Theorem 7. The complexity follows from Theorem 6. Indeed, since the algorithm has a polynomial complexity, also the size of Σ is polynomial. The correctness proof is based on the following invariant of the main repeat loop: the finite counter strategy $\pi = \Sigma^{\mathbf{r}}$ has these properties:

(a) Strategy π is $Safe_{\mathcal{M}}(s)$-safe in every state $s \in S$; in particular, we have for $l = \min\{\mathbf{r}(s), cap(\mathcal{M})\}$ that $RL_l(\alpha) \neq \bot$ for every finite path α produced by π from s.
(b) For each state $s \in S$ such that $\mathbf{r}(s) \leq cap(\mathcal{M})$ there exists a π-compatible finite path $\alpha = s_1 a_1 s_2 \dots s_n$ such that $s_1 = s$ and $s_n \in T$ and such that "the resource level with initial load $\mathbf{r}(s)$ never decreases below $\mathbf{r}$ along α", which means that for each prefix $\alpha_{..i}$ of α it holds $RL_{\mathbf{r}(s)}(\alpha_{..i}) \geq \mathbf{r}(s_i)$.

The theorem then follows from this invariant (parts (a) and the first half of (b)) and from Theorem 6. We start with the following support invariant, which is easy to prove.

Lemma 6. *The inequality* $\mathbf{r} \geq \mathit{Safe}_{\mathcal{M}}$ *is an invariant of the main repeat-loop.*

Proving Part (a) of the Main Invariant. We use the following auxiliary lemma.

Lemma 7. *Assume that* Σ *is a counter selector such that for all* $s \in S$ *such that* $\mathit{Safe}(s) < \infty$*:*

(1.) $\mathit{Safe}(s) \in \mathit{dom}(\Sigma(s))$.
(2.) For all $x \in \mathit{dom}(\Sigma(s))$*, for* $a = \Sigma(s)(x)$ *and for all* $t \in \mathit{Succ}(s,a)$ *we have* $RL_x(\mathit{sat}) = d - C(s,a) \geq \mathit{Safe}(t)$ *where* $d = x$ *for* $s \notin R$ *and* $d = \mathit{cap}(\mathcal{M})$ *otherwise.*

Then for each vector $\mathbf{y} \geq \mathit{Safe}$ *the strategy* $\pi = \Sigma^{\mathbf{y}}$ *is* $\mathit{Safe}(s)$*-safe in every state* s*.*

Proof. Let s be a state such that $\mathbf{y}(s) < \infty$. It suffices to prove that for every π-compatible finite path α started in s it holds $\bot \neq RL_{\mathbf{y}(s)}(\alpha)$. We actually prove a stronger statement: $\bot \neq RL_{\mathbf{y}(s)}(\alpha) \geq \mathit{Safe}(\mathit{last}(\alpha))$. We proceed by induction on the length of α. If $\mathit{len}(\alpha) = 0$ we have $RL_{\mathbf{y}(s)}(\alpha) = \mathbf{y}(s) \geq \mathit{Safe}_{\mathcal{M}}(s) \geq 0$. Now let $\alpha = \beta \odot t_1 a t_2$ for some shorter path β with $\mathit{last}(\beta) = t_1$ and $a \in A$, $t_1, t_2 \in S$. By induction hypothesis, $l = RL_{\mathbf{y}(s)}(\beta) \geq \mathit{Safe}_{\mathcal{M}}(t_1)$, from which it follows that $\mathit{Safe}_{\mathcal{M}}(t_1) < \infty$. Due to (1.), it follows that there exists at least one $x \in \mathit{dom}(\Sigma(t_1))$ such that $x \leq l$. We select maximal x satisfying the inequality so that $a = \Sigma(t_1)(x)$. We have that $RL_{\mathbf{y}(s)}(\alpha) = RL_l(t_1 a t_2)$ by definition and from (2.) it follows that $\bot \neq RL_x(t_1 a t_2) \geq \mathit{Safe}(t_2) \geq 0$. All together, as $l \geq x$ we have that $RL_{\mathbf{y}(s)}(\alpha) \geq RL_x(t_1 a t_2) \geq \mathit{Safe}(t_2) \geq 0$. □

Now we prove the part (a) of the main invariant. We show that throughout the execution of Algorithm 3, Σ satisfies the assumptions of Lemma 7. Property (1.) is ensured by the initialization on line 3. The property (2.) holds upon first entry to the main loop by the definition of a safe action (Definition 6). Now assume that $\Sigma(s)(\mathbf{r}(s))$ is redefined on line 13, and let a be the action $\mathbf{a}(s)$.

We first handle the case when $s \notin R$. Since a was selected on line 8, from the definition of *SPR-Val* we have that there is $t \in \mathit{Succ}(s,a)$ such that after the loop iteration,

$$\mathbf{r}(s) = C(s,a) + \max\{\mathbf{r}_{old}(t), \mathit{Safe}(t') \mid t \neq t' \in \mathit{Succ}(s,a)\} \geq C(s,a) + \max_{t' \in \mathit{Succ}(s,a)} \mathit{Safe}_{\mathcal{M}}(t'), \tag{1}$$

the latter inequality following from Lemma 6. Satisfaction of property (2.) in s then follows immediately from the Eq. (1).

If $s \in R$, then (1) holds before the truncation on line 10, at which point $\mathbf{r}(s) < cap(\mathcal{M})$. Hence, $cap(\mathcal{M}) - C(s,a) \geq \max_{t \in Succ(s,a)} Safe_{\mathcal{M}}(t)$ as required by (2.). From Lemmas 6 and 7 it follows that $\Sigma^{\mathbf{r}}$ is $Safe_{\mathcal{M}}(s)$-safe in every state s. This finishes the proof of part (a) of the invariant.

Proving Part (b) of the Main Invariant. Clearly, (b) holds after initialization. Now assume that an iteration of the main repeat loop was performed. Denote by π_{old} the strategy $\Sigma^{\mathbf{r}_{old}}$ and by π the strategy $\Sigma^{\mathbf{r}}$. Let s be any state such that $\mathbf{r}(s) \leq cap(\mathcal{M})$. If $\mathbf{r}(s) = \mathbf{r}_{old}(s)$, then we claim that (b) follows directly from the induction hypothesis: indeed, we have that there is an s-initiated π_{old}-compatible path α ending in a target state s.t. the $\mathbf{r}_{old}(s)$-initiated resource level along α never drops $\mathbf{r}_{old}$, i.e. for each prefix β of α it holds $RL_{\mathbf{r}_{old}(s)}(\beta) \geq \mathbf{r}_{old}(last(\beta))$. But then β is also π-compatible, since for each state q, $\Sigma(q)$ was only redefined for values smaller than $\mathbf{r}_{old}(q)$.

The case when $\mathbf{r}(s) < \mathbf{r}_{old}(s)$ is treated similarly. As in the proof of part (a), denote by a the action $\mathbf{a}(s)$ assigned on line 13. There must be a state $t \in Succ(s,a)$ s.t. (1) holds before the truncation on line 10. In particular, for this t it holds $RL_{\mathbf{r}(s)}(sat) \geq \mathbf{r}_{old}(t)$. By induction hypothesis, there is a t-initiated π_{old}-compatible path β ending in T satisfying the conditions in (b). We put $\alpha = sat \odot \beta$. Clearly α is s-initiated and reaches T. Moreover, it is π-compatible. To see this, note that $\Sigma^{\mathbf{r}}(s)(\mathbf{r}(s)) = a$; moreover, the resource level after the first transition is $e(t) = RL_{\mathbf{r}(s)}(sat) \geq \mathbf{r}_{old}(t)$, and due to the assumed properties of β, the $\mathbf{r}_{old}(t)$-initiated resource level (with initial load $e(t)$) never decreases below $\mathbf{r}_{old}$ along β. Since Σ was only re-defined for values smaller than those given by the vector $\mathbf{r}_{old}$, π mimics π_{old} along β. Since $\mathbf{r} \leq \mathbf{r}_{old}$, we have that along α, the $\mathbf{r}(s)$-initiated resource level never decreases below $\mathbf{r}$. This finishes the proof of part (b) of the invariant and thus also the proof of Theorem 7. □

6 Büchi

This section proofs Theorem 1 which is the main theoretical result of the paper. The proof is broken down into the following steps.

(1.) We identify a largest set $R' \subseteq R$ of reload states such that from each $r \in R'$ we can reach R' again (in at least one step) while consuming at most *cap* resource units and restricting ourselves only to strategies that (i) avoid $R \setminus R'$ and (ii) guarantee positive reachability of T in $\mathcal{M}(R')$.

(2.) We show that $SafeB\ddot{u}chi_{\mathcal{M},T} = SafePR_{\mathcal{M}(R'),T}$ and that the corresponding strategy (computed by Algorithm 3) is also T-Büchi $SafeB\ddot{u}chi_{\mathcal{M},T}(s)$-safe for each $s \in S$.

Algorithm 4: Almost-sure Büchi reachability of T in $\mathcal{M}$.

Input: CMDP $\mathcal{M} = (S, A, \Delta, C, R, cap)$, target states $T \subseteq S$
Output: The largest set $Rel \subseteq R$ such that $SafePR_{\mathcal{M}(Rel),T}(r) \leq cap$ for all $r \in Rel$.

$Rel \leftarrow R$; $ToRemove \leftarrow \emptyset$;
repeat
  $Rel \leftarrow Rel \setminus ToRemove$;
  $(\mathbf{reach}, \Sigma) \leftarrow SafePR_{\mathcal{M}(Rel),T}$;
  $ToRemove \leftarrow \{r \in Rel \mid \mathbf{reach}(r) > cap\}$;
until $ToRemove = \emptyset$;
return $\mathbf{reach}, \Sigma$

Algorithm 4 solves (1.) in a similar fashion as Algorithm 2 handled safety. In each iteration, we declare as non-reloading all states from which positive reachability of T and safety within $\mathcal{M}(Rel)$ cannot be guaranteed. This is repeated until we reach a fixed point. The number of iterations is clearly bounded by $|R|$.

Theorem 8. *Let $\mathcal{M} = (S, A, \Delta, C, R, cap)$ be a CMDP and $T \subseteq S$ be a target set. Moreover, let R' be the contents of Rel upon termination of Algorithm 4 for the input $\mathcal{M}$ and T. Finally let $\mathbf{r}$ and Σ be the vector and the selector returned by Algorithm 3 for the input $\mathcal{M}$ and T. Then for every state s, the finite counter strategy $\sigma = \Sigma^{\mathbf{r}}$ is T-Büchi $\mathbf{r}(s)$-safe in s in both $\mathcal{M}(R')$ and $\mathcal{M}$. Moreover, the vector $\mathbf{r}$ is equal to $SafeB\ddot{u}chi_{\mathcal{M},T}$.*

Proof. We first show that σ is T-Büchi $\mathbf{r}(s)$-safe in $\mathcal{M}(R')$ for all $s \in S$ with $\mathbf{r}(s) \leq cap$. Clearly it is $\mathbf{r}(s)$-safe, so it remains to prove that T is visited infinitely often with probability 1. We know that upon every visit of a state $r \in R'$, σ guarantees a future visit to T with positive probability. As a matter of fact, since σ is a finite memory strategy, there is $\delta > 0$ such that upon every visit of some $r \in R'$, the probability of a future visit to T is at least δ. As $\mathcal{M}(R')$ is decreasing, every s-initiated σ-compatible run must visit the set R' infinitely many times. Hence, with probability 1 we reach T at least once. The argument can then be repeated from the first point of visit to T to show that with probability 1 ve visit T at least twice, three times, etc. *ad infinitum.* By the monotonicity of probability, $\mathbb{P}^{\sigma}_{\mathcal{M},s}(\mathsf{B\ddot{u}chi}_T) = 1$.

It remains to show that $\mathbf{r} \leq SafeB\ddot{u}chi_{\mathcal{M},T}$. Assume that there is a state $s \in S$ and a strategy σ' such that σ' is d-safe in s for some $d < \mathbf{r}(s) = SafePR_{\mathcal{M}(R'),T}(s)$. We show that this strategy is not T-Büchi d-safe in $\mathcal{M}$. If all σ'-compatible runs reach T, then there must be at least one history α produced by σ' that visits $r \in R \setminus R'$ before reaching T (otherwise $d \geq \mathbf{r}(s)$).

Then either (a) $SafePR_{\mathcal{M},T}(r) = \infty$, in which case any σ'-compatible extension of α avoids T; or (b) since $SafePR_{\mathcal{M}(R'),T}(r) > cap$, there must be an extension of α that visits, between the visit of r and T, another $r' \in R \setminus R'$ such that $r' \neq r$. We can then repeat the argument, eventually reaching the case (a) or running out of the resource, a contradiction with σ' being d-safe. □

We can finally proceed to prove Theorem 1.

Proof (of Theorem 1). The theorem follows immediately from Theorem 8 since we can (a) compute $SafeB\ddot{u}chi_{\mathcal{M},T}$ and the corresponding strategy σ_T in polynomial time (see Theorem 7 and Algorithm 4); (b) we can easily check whether $d \geq SafeB\ddot{u}chi_{\mathcal{M},T}(s)$, if yes, than σ_T is the desired strategy σ; and (c) represent σ_T in polynomial space as it is a finite counter strategy represented by a polynomial-size counter selector. □

7 Implementation and Case Studies

We implemented the presented algorithms in Python and released it as an open-source tool called *FiMDP (Fuel in MDP)* available at https://github.com/xblahoud/FiMDP. The docker artifact is available at https://hub.docker.com/r/xblahoud/fimdp and can be run without installation via the Binder project [50]. We investigate the practical behavior of our algorithms using two case studies: (1) An autonomous electric vehicle (AEV) routing problem in the streets of Manhattan modeled using realistic traffic and electric car energy consumption data, and (2) a multi-agent grid world model inspired by the Mars Helicopter Scout [8] to be deployed from the planned Mars 2020 rover. The first scenario demonstrates the utility of our algorithm for solving real-world problems [59], while the second scenario studies the algorithm's scalability limits.

The consumption-Büchi objective can be also solved by a naive approach that encodes the energy constraints in the state space of the MDP, and solves it using techniques for standard MDPs [33]. States of such an MDP are tuples (s, e) where s is a state of the input CMDP and e is the current level of energy. Naturally, all actions that would lead to states with $e < 0$ lead to a special sink state. The standard techniques rely on decomposition of the MDP into maximal end-components (MEC). We implemented the explicit encoding of CMDP into MDP, and the MEC-decomposition algorithm.

All computations presented in the following were performed on a PC with Intel Core i7-8700 3.20 GHz 12 core processor and a RAM of 16 GB running Ubuntu 18.04 LTS. All running times are means from at least 5 runs and the standard deviation was always below 5% among these runs.

7.1 Electric Vehicle Routing

We consider the area in the middle of Manhattan, from 42nd to 116th Street, see Fig. 2. Street intersections and directions of feasible movement form the state and action spaces of the MDP. Intersections in the proximity of real-world fast charging stations [56] represent the set of reload states.

After the AEV picks a direction, it reaches the next intersection in that direction deterministically with a stochastic energy consumption. We base our model of consumption on distributions of vehicle travel times from the area [55] and conversion of velocity and travel times to energy consumption [52]. We discretize the consumption distribution into three possible values (c_1, c_2, c_3) reached with corresponding probabilities (p_1, p_2, p_3). We then model the transition from one intersection (I_1) to another (I_2) using additional dummy states as explained in Fig. 2.

The corresponding CMDP has 7378 states and 8473 actions. For a fixed set of 100 randomly selected target states, Fig. 3 shows influence of requested capacity on running times for **(a)** strategy for Büchi objective using CMDP (our approach), and **(b)** MEC-decomposition for the corresponding explicit MDP. With constant number of states, our algorithm runs rea-

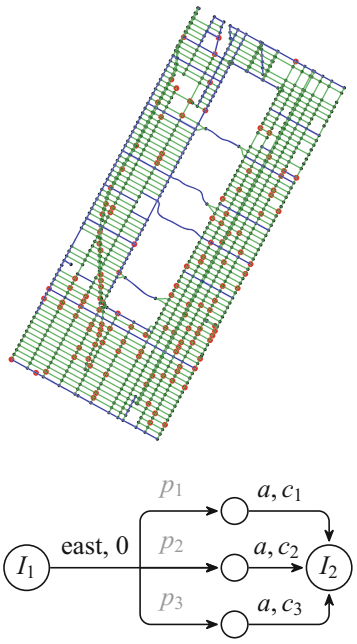


Fig. 2. (Top:) Street network in the considered area. Charging stations are red, one way roads green, and two-way roads blue. (Bottom:) Transition from intersection I_1 to I_2 with stochastic consumption. The small circles are dummy states. (Color figure online)

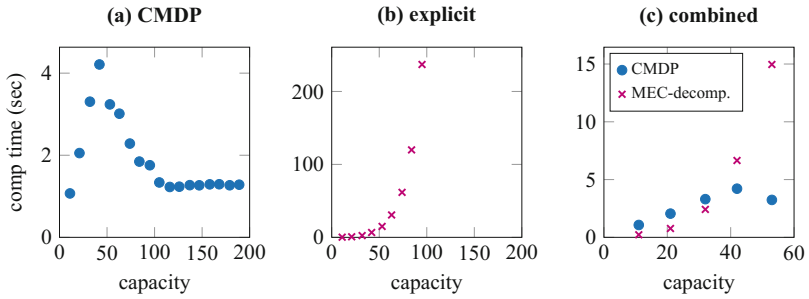


Fig. 3. Mean computation times for a fixed target set of size 100 and varying capacity: **(a) CMDP** – computating Büchi objective via CMDP, **(b) explicit** – computating MEC decomposition of the explicit MDP, **(c) combined** – **(a)** and **(b)** combined for small capacity values.

sonably fast for all capacities and the running time stabilizes for $cap > 95$; this is not the case for the explicit approach where the number of states keeps growing (52747 for $cap = 95$) as well as the running time. The decomposition to MECs is slightly faster than solving Büchi using CMDP for the small capacities (Fig. 3 (c)), but MECs decomposition is only a part of the solution and running the full algorithm for Büchi would most likely diminish this advantage.

7.2 Multi-agent Grid World

We use multi-agent grid world to generate CMDP with huge number of states to study the scalability limits of the proposed algorithms. We model the rover and the helicopter of the Mars 2020 mission with the following realistic considerations: the rover enjoys infinite energy while the helicopter is restricted by batteries recharged at the rover. These two vehicle jointly operate on a mission where the helicopter reaches areas inaccessible to the rover. The outcomes of the helicopter's actions are deterministic while those of the rover—influenced by terrain dynamics—are stochastic. For a grid world of size n, this system can be naturally modeled as a CMDP with n^4 states. Figure 4 shows the running times of the Büchi objective for growing grid sizes and capacities in CMDP. We observe that the increase in the computational time of CMDP follows the growth in the number of states roughly linearly, and our implementation deals with an MDP with 1.6×10^5 states in no more than seven minutes. The figure also shows the running time for the MEC decomposition of the corresponding explicit MDP when the capacity is 10 and, for certain smaller, computationally feasible grid sizes, when the capacity is 20.

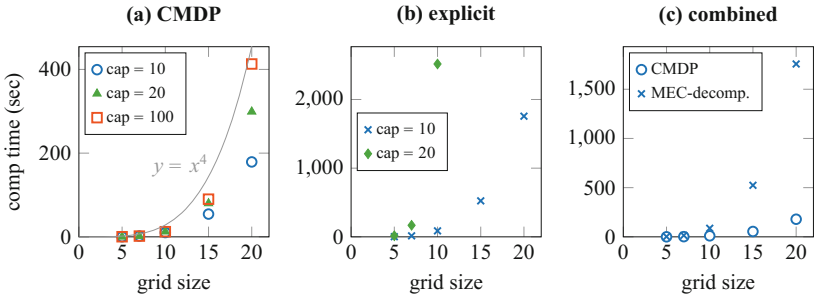


Fig. 4. Mean computation times for varying grid sizes and of size capacities: **(a) CMDP** – computating Büchi objective via CMDP, the gray line shows the corresponding growth in the number of states on separate scale, **(b) explicit** – computating MEC decomposition of the explicit MDP, **(c) combined** – combined computation time for a capacity of 10.

8 Conclusion and Future Work

We presented a first study of consumption Markov decision processes (CMDPs) with qualitative ω-regular objectives. We developed and implemented a polynomial-time algorithm for CMDPs with an objective of probability-1 satisfaction of a given Büchi condition. Possible directions for the future work are extensions to quantitative analysis (e.g. minimizing the expected resource consumption), stochastic games, or partially observable setting.

Acknowledgements. We acknowledge the kind help of Vojtěch Forejt, David Klaška, and Martin Kučera in the discussions leading to this paper.

References

1. Abdulla, P.A., Atig, M.F., Hofman, P., Mayr, R., Kumar, K.N., Totzke, P.: Infinite-state energy games. In: Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 7:1–7:10 (2014)
2. Ash, R., Doléans-Dade, C.: Probability and Measure Theory. Harcourt/Academic Press, San Diego (2000)
3. Bacci, G., Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Reynier, P.-A.: Optimal and robust controller synthesis. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 203–221. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_12
4. Baier, C., Chrszon, P., Dubslaff, C., Klein, J., Klüppelholz, S.: Energy-utility analysis of probabilistic systems with exogenous coordination. In: de Boer, F., Bonsangue, M., Rutten, J. (eds.) It's All About Coordination. LNCS, vol. 10865, pp. 38–56. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90089-6_3
5. Baier, C., Daum, M., Dubslaff, C., Klein, J., Klüppelholz, S.: Energy-utility quantiles. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 285–299. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_24
6. Baier, C., Dubslaff, C., Klein, J., Klüppelholz, S., Wunderlich, S.: Probabilistic model checking for energy-utility analysis. In: van Breugel, F., Kashefi, E., Palamidessi, C., Rutten, J. (eds.) Horizons of the Mind. A Tribute to Prakash Panangaden. LNCS, vol. 8464, pp. 96–123. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06880-0_5
7. Baier, C., Dubslaff, C., Klüppelholz, S., Leuschner, L.: Energy-utility analysis for resilient systems using probabilistic model checking. In: Ciardo, G., Kindler, E. (eds.) PETRI NETS 2014. LNCS, vol. 8489, pp. 20–39. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07734-5_2
8. Balaram, B., et al.: Mars helicopter technology demonstrator. In: AIAA Atmospheric Flight Mechanics Conference (2018)
9. Boker, U., Henzinger, T.A., Radhakrishna, A.: Battery transition systems. In: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 595–606 (2014)
10. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed automata with observers under energy constraints. In: 13th ACM International Conference on Hybrid Systems: Computation and Control, pp. 61–70. ACM (2010)

11. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite runs in weighted timed automata with energy constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85778-5_4
12. Bouyer, P., Hofman, P., Markey, N., Randour, M., Zimmermann, M.: Bounding average-energy games. In: Esparza, J., Murawski, A.S. (eds.) FoSSaCS 2017. LNCS, vol. 10203, pp. 179–195. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54458-7_11
13. Bouyer, P., Markey, N., Randour, M., Larsen, K.G., Laursen, S.: Average-energy games. Acta Informatica **55**(2), 91–127 (2018)
14. Brázdil, T., Chatterjee, K., Kučera, A., Novotný, P.: Efficient controller synthesis for consumption games with multiple resource types. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 23–38. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_8
15. Brázdil, T., Jančar, P., Kučera, A.: Reachability games on extended vector addition systems with states. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14162-1_40
16. Brázdil, T., Klaška, D., Kučera, A., Novotný, P.: Minimizing running costs in consumption systems. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 457–472. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_30
17. Brázdil, T., Kučera, A., Novotný, P.: Optimizing the expected mean payoff in energy Markov decision processes. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 32–49. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_3
18. Brenguier, R., Cassez, F., Raskin, J.-F.: Energy and mean-payoff timed games. In: 17th International Conference on Hybrid Systems: Computation and Control, pp. 283–292 (2014)
19. Brihaye, T., Geeraerts, G., Haddad, A., Monmege, B.: Pseudopolynomial iterative algorithm to solve total-payoff games and min-cost reachability games. Acta Informatica **54**(1), 85–125 (2017)
20. Brim, L., Chaloupka, J., Doyen, L., Gentilini, R., Raskin, J.: Faster algorithms for mean-payoff games. Form. Methods Syst. Des. **38**(2), 97–118 (2011)
21. Bruyère, V., Hautem, Q., Randour, M., Raskin, J.-F.: Energy mean-payoff games. In: 30th International Conference on Concurrency Theory, pp. 21:1–21:17 (2019)
22. Cachera, D., Fahrenberg, U., Legay, A.: An ω-algebra for real-time energy problems. Log. Methods Comput. Sci. **15**(2) (2019)
23. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45212-6_9
24. Chaloupka, J.: Z-reachability problem for games on 2-dimensional vector addition systems with states is in P. Fundamenta Informaticae **123**(1), 15–42 (2013)
25. Chatterjee, K.: Stochastic ω-regular games. Ph.D. thesis, University of California, Berkeley (2007)
26. Chatterjee, K., Doyen, L.: Energy and mean-payoff parity Markov decision processes. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 206–218. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22993-0_21
27. Chatterjee, K., Doyen, L.: Energy parity games. Theor. Comput. Sci. **458**, 49–60 (2012)

28. Chatterjee, K., Doyen, L., Henzinger, T., Raskin, J.-F.: Generalized mean-payoff and energy games. In: 30th Annual Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 505–516 (2010)
29. Chatterjee, K., Henzinger, M., Krinninger, S., Nanongkai, D.: Polynomial-time algorithms for energy games with special weight structures. In: 20th Annual European Symposium on Algorithms, pp. 301–312 (2012)
30. Chatterjee, K., Jurdziński, M., Henzinger, T.: Quantitative stochastic parity games. In: 15th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 121–130 (2004)
31. Chatterjee, K., Randour, M., Raskin, J.-F.: Strategy synthesis for multi-dimensional quantitative objectives. Acta informatica **51**(3–4), 129–163 (2014)
32. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995)
33. de Alfaro, L.: Formal verification of probabilistic systems. Ph.D. thesis, Stanford University (1998)
34. Degorre, A., Doyen, L., Gentilini, R., Raskin, J.-F., Toruńczyk, S.: Energy and mean-payoff games with imperfect information. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 260–274. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15205-4_22
35. Ésik, Z., Fahrenberg, U., Legay, A., Quaas, K.: An algebraic approach to energy problems I - continuous Kleene ω-algebras. Acta Cybernetica **23**(1), 203–228 (2017)
36. Ésik, Z., Fahrenberg, U., Legay, A., Quaas, K.: An algebraic approach to energy problems II - the algebra of energy functions. Acta Cybernetica **23**(1), 229–268 (2017)
37. Fahrenberg, U., Juhl, L., Larsen, K.G., Srba, J.: Energy games in multiweighted automata. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 95–115. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23283-1_9
38. Fahrenberg, U., Legay, A.: Featured weighted automata. In: 5th International FME Workshop on Formal Methods in Software Engineering, pp. 51–57 (2017)
39. Fijalkow, N., Zimmermann, M.: Cost-parity and cost-Streett games. In: 32nd Annual Conference on Foundations of Software Technology and Theoretical Computer Science, pp. 124–135 (2012)
40. Filiot, E., Gentilini, R., Raskin, J.-F.: Quantitative languages defined by functional automata. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 132–146. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_11
41. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Good-for-MDPs automata for probabilistic analysis and reinforcement learning. TACAS 2020. LNCS, vol. 12078, pp. 306–323. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_17
42. Herrmann, L., Baier, C., Fetzer, C., Klüppelholz, S., Napierkowski, M.: Formal parameter synthesis for energy-utility-optimal fault tolerance. In: Bakhshi, R., Ballarini, P., Barbot, B., Castel-Taleb, H., Remke, A. (eds.) EPEW 2018. LNCS, vol. 11178, pp. 78–93. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02227-3_6
43. Juhl, L., Guldstrand Larsen, K., Raskin, J.-F.: Optimal bounds for multiweighted and parametrised energy games. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 244–255. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_15

44. Jurdziński, M., Lazić, R., Schmitz, S.: Fixed-dimensional energy games are in pseudo-polynomial time. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 260–272. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47666-6_21
45. Jurdziński, M.: Deciding the winner in parity games is in UP ∩ co-UP. Inf. Process. Lett. **68**(3), 119–124 (1998)
46. Khachiyan, L., et al.: On short paths interdiction problems: total and node-wise limited interdiction. Theory Comput. Syst. **43**(2), 204–233 (2008)
47. Klaška, D.: Complexity of Consumption Games. Bachelor's thesis, Masaryk University (2014)
48. Larsen, K.G., Laursen, S., Zimmermann, M.: Limit your consumption! Finding bounds in average-energy games. In: 14th International Workshop Quantitative Aspects of Programming Languages and Systems, pp. 1–14 (2016)
49. Mayr, R., Schewe, S., Totzke, P., Wojtczak, D.: MDPs with energy-parity objectives. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 1–12 (2017)
50. Jupyter, P., et al.: Binder 2.0 - reproducible, interactive, sharable environments for science at scale. In: 17th Python in Science Conference, pp. 113–120 (2018)
51. Sickert, S., Esparza, J., Jaax, S., Křetínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 312–332. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_17
52. Straubel, J.B.: Roadster efficiency and range (2008). https://www.tesla.com/blog/roadster-efficiency-and-range
53. Sugumar, G., Selvamuthukumaran, R., Dragicevic, T., Nyman, U., Larsen, K.G., Blaabjerg, F.: Formal validation of supervisory energy management systems for microgrids. In: 43rd Annual Conference of the IEEE Industrial Electronics Society, pp. 1154–1159 (2017)
54. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
55. Uber Movement: Traffic speed data for New York City (2019). https://movement.uber.com/
56. United States Department of Energy. Alternative fuels data center (2019). https://afdc.energy.gov/stations/
57. Velner, Y., Chatterjee, K., Doyen, L., Henzinger, T.A., Rabinovich, A.M., Raskin, J.: The complexity of multi-mean-payoff and multi-energy games. Inf. Comput. **241**, 177–196 (2015)
58. Wognsen, E.R., Hansen, R.R., Larsen, K.G., Koch, P.: Energy-aware scheduling of FIR filter structures using a timed automata model. In: 19th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, pp. 1–6 (2016)
59. Zhang, H., Sheppard, C.J.R., Lipman, T.E., Moura, S.J.: Joint fleet sizing and charging system planning for autonomous electric vehicles. IEEE Trans. Intell. Transp. Syst. (2019)

STMC: Statistical Model Checker with Stratified and Antithetic Sampling

Nima Roohi[1(✉)], Yu Wang[2], Matthew West[3], Geir E. Dullerud[3], and Mahesh Viswanathan[3]

[1] University of California, San Diego, USA
nroohi@ucsd.edu
[2] Duke University, Durham, USA
yw354@duke.edu
[3] University of Illinois at Urbana-Champaign, Urbana, USA
{mwest,dullerud,vmahesh}@illinois.edu

Abstract. `STMC` is a statistical model checker that uses antithetic and stratified sampling techniques to reduce the number of samples and, hence, the amount of time required before making a decision. The tool is capable of statistically verifying any *black-box* probabilistic system that `PRISM` can simulate, against probabilistic bounds on any property that `PRISM` can evaluate over individual executions of the system. We have evaluated our tool on many examples and compared it with both symbolic and statistical algorithms. When the number of strata is large, our algorithms reduced the number of samples more than 3 times on average. Furthermore, being a statistical model checker makes `STMC` able to verify models that are well beyond the reach of current symbolic model checkers. On large systems (up to 10^{14} states) `STMC` was able to check 100% of benchmark systems, compared to existing symbolic methods in `PRISM`, which only succeeded on 13% of systems. The tool, installation instructions, benchmarks, and scripts for running the benchmarks are all available online as open source.

1 Introduction

Statistical model checking (SMC) plays an important role in verifying probabilistic temporal logics on cyber-physical systems [1,14,15]. In SMC, we treat the objective bounded temporal specifications as statistical hypothesis, and infer their correctness with high confidence from samples of the systems. Compared to analytic approaches, statistical model checkers rely only on samples from the systems, and hence are more scalable to large real-world problems with complicated stochastic behavior [3,6,18].

To our knowledge, all existing SMC tools use independent samples. Admittedly, independent sampling is easy to implement, and it is the only option when the model is completely unknown. However, as shown recently in [24,25], if the model is partially known, then we can exploit this knowledge to generate

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 448–460, 2020.
https://doi.org/10.1007/978-3-030-53291-8_23

semantically negatively correlated samples to increase the sample efficiency in SMC. In [24,25], we present the *stratified and antithetic sampling* techniques for discrete-time Markov chains (DTMC). In this work, we extend the technique to continuous-time Markov chains (CTMC), and implement the corresponding SMC algorithms in the tool `STMC`. The tool is evaluated on several case studies under hundreds of different scenarios, some of which are well beyond the capabilities of current symbolic model checkers. The results show that the sample efficiency can be significantly improved by using semantically negatively correlated sampling, instead of independent sampling.

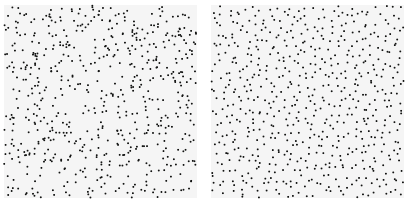
This work also provides experimental comparisons between our SMC method and common symbolic model checking methods. Since we use large values for parameters in our case studies, it is no surprise that symbolic engines fail on many of them. However, without our results, the meaning of the word "large" is unclear. Our results give a good understanding of what is currently beyond the capabilities of symbolic engines in a popular tool like `PRISM`. Next, restricting our attention to the cases in which symbolic engines successfully terminate, our results give us a helpful comparison between symbolic and statistical verification times. It is well-known that symbolic algorithms do not scale well, while statistical ones do. However, that knowledge alone does not give us any insight into how much more or less time a symbolic method requires compared to a statistical one. Finally, when a symbolic method terminates, one might argue that its result is far more valuable than the result of a statistical approach since statistical methods can produce incorrect results. Unfortunately, that is not entirely true. Since the complexity of solving a problem is too high in practice, many symbolic algorithms, including those in `PRISM`, employ an iterative method to approximate probabilities. This approximation can be far from the actual probability, leading to incorrect model checking results (*e.g.,* [5]).

Related Work. Among the existing statistical model checkers, `PRISM` [4,12], `MRMC` [10], `VESTA` [19], `YMER` [27], and `COSMOS` [2] only support independent sampling on DTMC, CTMC, or other more general probabilistic models. `PLASMA` [9] also supports importance sampling. In importance sampling, although samples may have different weights, they are still generated independently. To our knowledge, our tool `STMC` is the only existing statistical model checker that employs semantically negatively related sampling on DTMC and CTMC.

2 Stratified and Antithetic Sampling

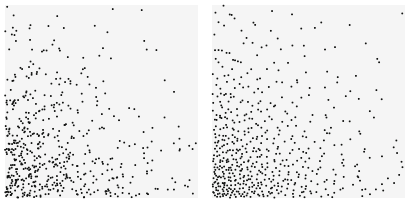
Stratified and antithetic samplings are two approaches for generating negatively correlated random samples. When using stratified sampling to draw n samples from a distribution, we divide the support into sets with equal measure, and then draw one sample from each partition. When using antithetic sampling, a random seed is first drawn from $x \in [0, 1]$, and then two correlated samples are generated using x and $1 - x$, respectively. Figures 1 and 2 compare independent and stratified sampling for 625 samples that we drew from the joint distribution

of two random variables. In Fig. 1, each variable is uniformly distributed in $[0, 1]$, and in Fig. 2, each variable is exponentially distributed with rate 3 (we only show samples that are within the unit square). It is clear that the stratified samples are (visually) better distributed in both figures.



(a) Independent **(b)** Stratified

Fig. 1. Uniform distribution



(a) Independent **(b)** Stratified

Fig. 2. Exponential distribution

We have shown in [24] that by choosing a proper representation of a Markov chain, the stratified sampling technique can be applied to generate semantically negatively correlated sample paths. This technique reduces the sampling cost for statistically verifying temporal formulas. In the rest of this section, we list two algorithms: Stratified sampling of a CTMC, and stratified sequential probability ratio test for a CTMC. The antithetic variants are simpler and we do not present them here for the lack of space. Compared to our algorithms in [24], there are two main differences. First, we present these algorithms for CTMCs instead of DTMCs, as they are slightly more involved. Second, for the stratified sampling of a CTMC, our algorithm supports stratification over multiple steps directly.

Algorithm 1 shows the pseudo-code for stratified sampling of a CTMC; to obtain a stratified sampling algorithm for DTMC, we only need to remove π_2, index_2, offset_2, rate, r_2, and r_3. It takes two inputs: ψ, a temporal formula that we want to evaluate on every sampled path, and strata_sizes , the number of strata at every step. This is a non-empty list of positive integers. Let K be the length of this list, and N be the product of its elements. If the i_{th} item of the list is n then the number of strata at steps $i, i + K, i + 2K, i + 3K, \ldots$ must be n.[1] The algorithm simultaneously simulates N paths and terminates after the value of ψ on all these paths are known. Inside the main loop, simulation is performed incrementally, K steps at a time. Random permutations π_1, π_2, and variables index_1, index_2 are used to make simulations of every K steps and random numbers r_1 and r_2 (defined later in the code) independent of each other. The number of strata at every step is an input to this algorithm. Using that number, variables offset_1 and offset_2 determine which strata we should use at step s. Finally, r_2 is a uniformly distributed stratified sample in $[0, 1)$. However,

[1] The current version of `PRISM` only handles one initial state for simulation. Therefore, there will be no stratification for initializing paths.

we need an exponentially distributed stratified sample, which is precisely what $-\ln(1-r_2)/\text{rate}$ gives us.

Algorithm 2 shows pseudo-code for statistical verification of CTMC and DTMC using stratified samples. The algorithm is quite simple. It keeps sampling using Algorithm 1 and computes the average and variance of the values it receives until a termination condition is satisfied. Checking the termination conditions after every step suggests using an online algorithm for computing the mean and variance of samples. We use Welford's online algorithm [26] in our implementation.

Algorithm 1 Stratified Sampling for CTMC

```
// Take stratified samples and return fraction of samples that satisfy ψ.
// Param ψ is an LTL formula.
// Param strata_sizes is a non−empty list of positive integers.
function stratified_sampling(ψ, strata_sizes)
  val K     = strata_sizes.length      // Length of the list
  val N     = strata_sizes.product     // Product of elements in the list
  val paths = initialize N paths       // index starts at 0
  val evals = initialize N evaluators // incrementally evaluate ψ on paths
  // Evaluation in the condition of the while loop is performed by PRISM
  while(∃ j∈{0,...,N−1}, evals[j](path[j])='unknown')
    val π1 = random permutation of 0,1,...,N−1
    val π2 = random permutation of 0,1,...,N−1
    for(i ← 0,...,N−1)
      vars index1, index2 = π1[i], π2[i]
      for(s ← 0,...,K−1)
        val  size = strata_sizes[s]  // number of strata at step s
        vals offset1, offset2 = index1%size, index2%size
        index1, index2 /= size
        val rate = rate of last state in path[i]    // by PRISM
        val r1   = rnd(0,1) / size + offset1 / size // rnd(0,1) ∈ [0,1)
        val r2   = rnd(0,1) / size + offset2 / size
        val r3   = −ln(1−r2) / rate // stratified exponentially distributed
        Simulate one step in path[i] using r1 and r3 // by PRISM
  return number of paths that satisfy ψ / N
```

Finally, one can extend the following results from [24] to include CTMC.

Theorem 1. *Let ψ be a bounded LTL formula.*

1. *The output of Algorithm 1 has the same expected value as the probability of a random path satisfying ψ.*
2. *If ψ is of the form $\psi_1 \mathcal{U}_I \psi_2$, such that the set of states satisfying ψ_2 is a subset of the same set for ψ_1, then the satisfaction values of different paths simulated by Algorithm 1 are non-positively correlated.*

Theorem 2. *The sampling cost of Algorithm 2 is asymptotically no more than the sampling cost of SPRT [20] using i.i.d. samples.*

3 Tool Architecture

We have implemented our algorithms in `Scala` and published it under the GNU General Public License v3.0. The tool can be downloaded from https://github.com/nima-roohi/STMC/, where installation instructions, benchmarks, and scripts for running the benchmarks are located. We use `PRISM` to load models from files, simulate them, and evaluate simulated paths against non-probabilistic bounded temporal properties. Therefore, `STMC` is capable of statistically verifying any model, as long as it can be simulated by `PRISM`, and bounded temporal properties can be evaluated on single executions of that model. Figure 3 shows `STMC` at a very high level. Boxes marked with 'P' are where we directly use `PRISM`.

Algorithm 2 Stratified Sequential Probability Ratio Test

```
1  // Verify P_{≤t} ψ using stratified sampling.
2  // Param t is the input threshold
3  // Param ψ is an LTL formula (non−probabilistic).
4  // Param strata_sizes is a non−empty list of positive integers.
5  // Param min_iter is the minimum number of iters. the algorithm should take.
6  // Param α is Type−I error probability (must satisfy 0 < α < 1/2).
7  // Param β is Type−II error probability (must satisfy 0 < β < 1/2).
8  // Param δ is half of the size of indifference region.
9  function stratified_SPRT(ℙ_{≤t} ψ, strata_sizes, min_iter, α, β, δ)
10   var iter = 1
11   var μ    = 0 // average of stratified_sampling return values
12   var σ    = 0 // standard deviation of stratified_sampling return values
13   while(true)
14     iter++
15     val x = stratified_sampling(ψ,strata_sizes)
16     update μ and σ using x // e.g. Welford's online algorithm [27]
17     if iter > min_iter then
18       if μ − t < −σ²/(2δ iter) ln (1−α)/β then return true // accept P_{≤t} ψ
19       if μ − t >  σ²/(2δ iter) ln (1−β)/α then return false // reject P_{≤t} ψ
```

Executions of `STMC` are configured through different options/switches. The most basic options are `help`, which prints out a list of switches for both `STMC` and `PRISM`, and `stmc`, which enables the tool (without `stmc`, everything will be passed to `PRISM`, pretty much like `STMC` was not there in the first place). Statistical verification is enabled using option `sim`; it is always required when `stmc` is used. The sampling method is specified using option `smp_method` or `sm`. Possible values for the sampling method are `independent`, `antithetic`, and `stratified`. Using option `hyp_test_method` or `hm`, users also have to specify a hypothesis testing method that they would like to use. Supported values for this option are currently `SPRT`, `TSPRT`, `GLRT`, and `SSPRT`. `SPRT` is used for the sequential probability ratio test [20]. This algorithm has already been implemented in `PRISM` and in our experience it has a very similar performance to our implementation (`SPRT` in Sect. 4 refers to the implementation from `PRISM`). We use our implementation for the next option, `TSPRT`. Sequential probability ratio test assumes that the actual probability is not within the δ-neighborhood of the input

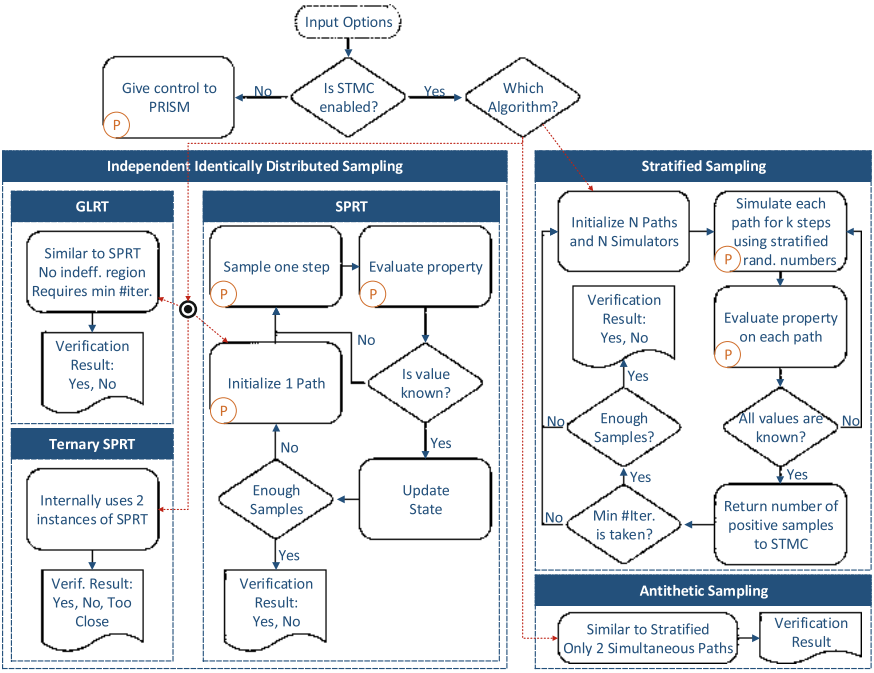


Fig. 3. Architecture of `STMC`. Boxes marked with letter 'P' use `PRISM` directly. N is the number of strata, K is the length of strata-size list (see option `strata_size` below).

threshold. If this assumption is not satisfied, then the algorithm does not guarantee any error probability. `TSPRT`, which stands for Ternary SPRT, solves this problem by introducing a third possible answer: `TOO_CLOSE`. The algorithm was introduced in [28]. *Without* assuming that the actual probability is not within the δ-neighborhood of the input threshold, `TSPRT` guarantees Type-I and Type-II error probabilities are bounded by the input parameters α and β, respectively. Furthermore, it guarantees that if the actual probability and the input threshold are not δ-close, then the probability of returning `TOO_CLOSE` is less than another input parameter γ; we call this Type-III error probability. The sequential probability ratio test was originally developed for simple hypotheses, and the test is not necessarily optimal when composite hypotheses are used [13]. To overcome this problem, the generalized likelihood ratio test (GLRT) was designed in [7]. The algorithm does not require an indifference region as an input parameter and provides guarantees on Type-I and Type-II error probabilities *asymptotically*. The main issue with this test is that since probabilistic error guarantees are asymptotic, for the test to perform reasonably well in practice (*i.e.*, respect the input error parameters), a correct minimum number of samples must be given as an extra input parameter. If this parameter is too large then the number of samples will be unnecessarily high, and if the parameter is too small then the actual error probability of the algorithm could be close to 0.5, even though

the input error parameters are set to, for example, 10^{-7}. The last possible value for `hyp_test_method` is `SSPRT`, which stands for Stratified SPRT. This option is used whenever stratified or antithetic samplings are desired.

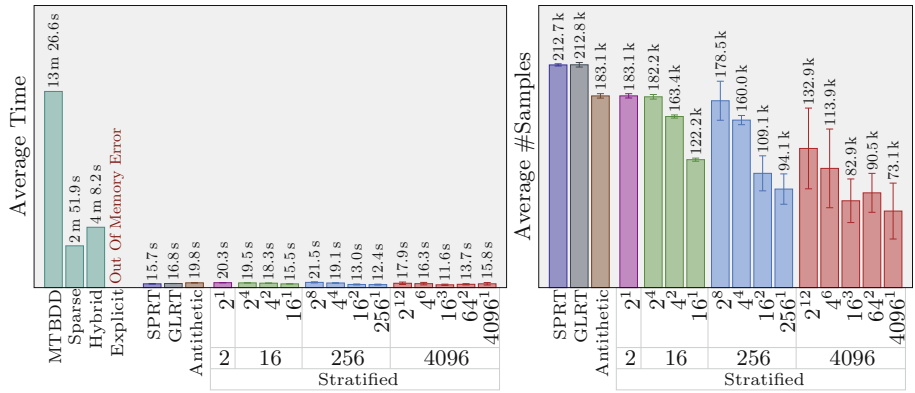
When stratification is used, the number of strata should be specified using option `strata_size` or `ss`. It is a comma-separated list of positive integers. For example, 4, 4, 4, 4, 4, 4 specifies 4 strata for six consecutive steps (4096 total), and 4096 specifies 4096 strata for every single step. Note that in both of these examples, stratified sampling simultaneously takes 4096 sample paths, which requires more memory. However, we saw in our experiments that for non-nested temporal formulas, at most two states of each path are stored into memory. Therefore, even larger strata sizes should be possible. This was the most challenging part of the implementation, because the simulator engine in `PRISM` is written assuming that paths are sampled one by one. However, if we followed the same approach in `STMC`, we would have to store every random number that was previously generated, which increased the amount of memory used for simulation from $\mathcal{O}(1)$ to $\mathcal{O}(N \times L)$, where N is the number of strata and L is the maximum length of simulated paths. By simulating the paths simultaneously, we only use $\mathcal{O}(N)$ bytes of memory. Next, Type-I, Type-II, Type-III, and half of the size of the indifference region are specified using `alpha`, `beta`,[2] `gamma` and `delta`, respectively (not every algorithm uses all of these parameters). Finally, most algorithms that use variance in their termination condition, require help when sample variance remains zero after the first few iterations. `STMC` uses `min_iter` for this purpose, and `PRISM` uses `simvar`.

4 Experimental Results

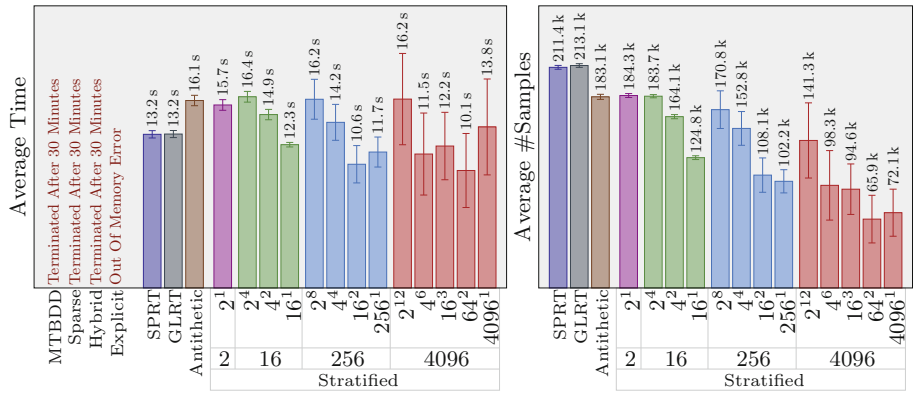
We evaluated our algorithms on 10 different sets of examples. Each set contains four variations of the same problem with varying parameters and, hence, various sizes, and each of those variations includes four symbolic tests as well as 16 statistical ones. Furthermore, we repeat each of the statistical tests 20 times, to compute 95% confidence intervals for time and number of samples taken by the statistical algorithms. This gives us a total of 800 tests and 12 960 runs to obtain results for those tests. Regarding the stratified sampling, for each variation, we consider 13 settings in 4 groups. Each group uses a different number of strata: 2, 16, 256, and 4096. When the number of strata is more than 2, we also consider different possibilities for how to divide strata among different steps. For example, when 256 strata are used, 256^1 means every step has 256 strata, but different steps are independent of each other. On the other hand, 2^8 means every step has only two strata, but stratification is performed over every 8 consecutive steps.

For the sake of space, we only present 15% of our results in this paper. Full experimental results are available at https://nima-roohi.github.io/STMC/#/benchmarks. Also, all the benchmark source files, along with scripts for running them, can be obtained from the tool's repository page https://github.com/nima-roohi/STMC/. The parameters we chose resulted in large systems, and

[2] To the best of our knowledge, `PRISM` always assumes $\alpha = \beta$.



(a) N: 90, K: 5, States: 113 384 792, Transitions: 180 005 807

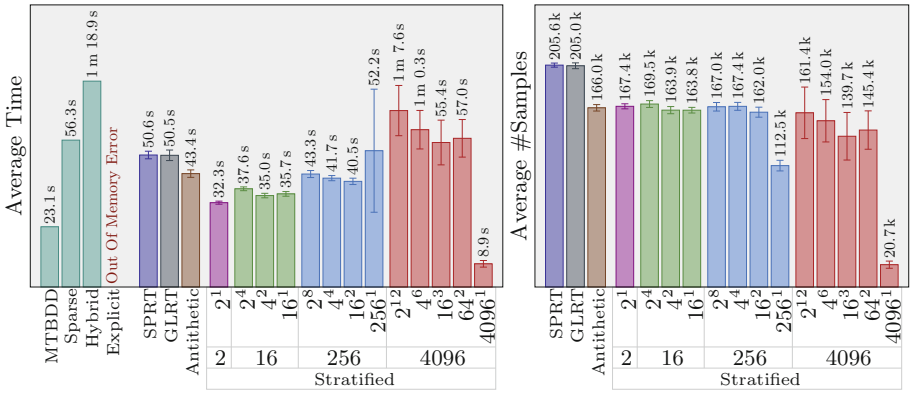


(b) N: 150, K: 11, States: 1 849 234 352, Transitions: 2 944 935 077

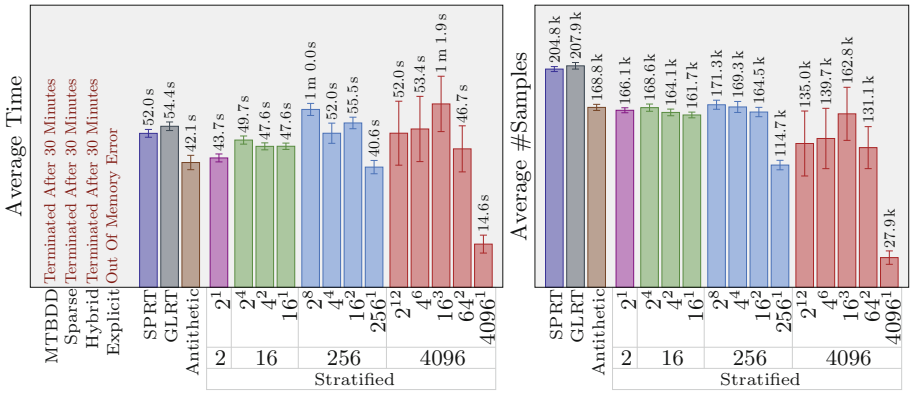
Fig. 4. NAND multiplexing (DTMC - macOS) [17]

significant time has been spent to run and collect the results. To perform our experiments faster, we ran all of our tests using four processes (using option '`-mt 4`'). We also divided out our 10 sets of examples into two groups and ran each set on one of two machines. One of them is running Ubuntu 18.04 with an i7-8700 CPU 3.2 GHz and 16 GB memory, and the other one is running macOS Mojave with an i7 CPU 3.5 GHz and 32 GB memory. `STMC`'s webpage contains a short description for each example and a link to another page for the full explanation. We end this section with a few notes regarding our results.

1. Like any statistical test that is run in a black-box setting, we need to assume simulation of every path will eventually terminate. In fact, `PRISM` uses the parameter `simpathlen`, with 10 000 as its default value, to restrict the maximum number of simulation steps in each path. Currently, `simpathlen` can be as large as $2^{63} - 1$, which is more than enough in most practical applications.



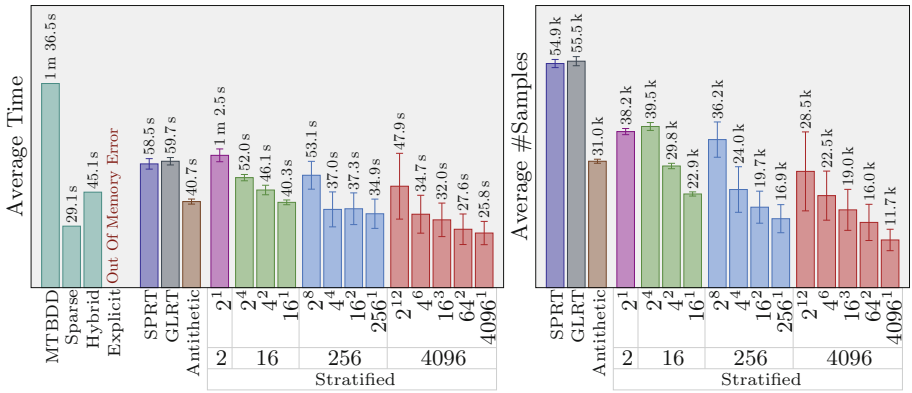
(a) `MAX_COUNT`: 10 000, States: 8 451 788, Transitions: 35 677 505



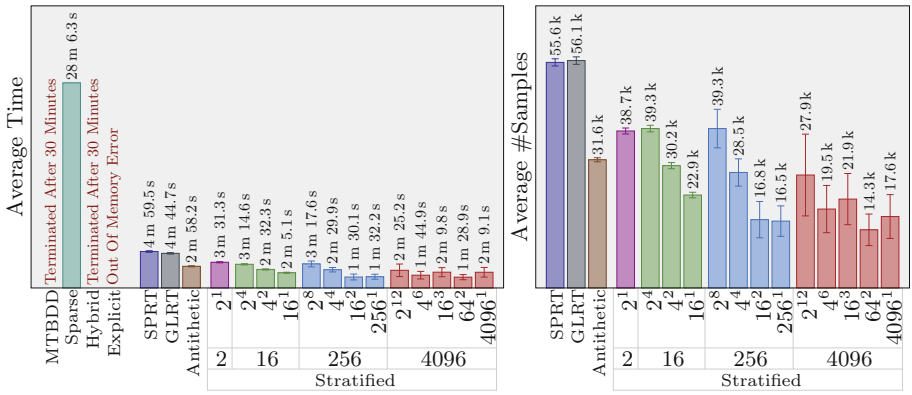
(b) `MAX_COUNT`: 1 000 000, States: about 845 017 880, Transitions: about 3 567 075 050

Fig. 5. Embedded control system (CTMC - Ubuntu) [11,16]

2. To make the configurations less in favor of statistical algorithms, we used small values for α, β, and δ in our benchmarks (between 0.0001 and 0.001). Also, we have estimated the actual probabilities using a symbolic model checker or using a statistical algorithm in `PRISM` and set the threshold close to the actual probability. These settings cause the statistical algorithms to take more samples, which indeed makes it possible for us to observe the effect of antithetic and stratification on the number of samples. As a side effect, we did not observe any performance benefits of GLRT over SPRT.
3. In many of our examples, the variance is particularly high when strata size is 4096. This is because in our benchmarks, whenever 4096 strata are used, we set the minimum number of iterations to 2 (*i.e.,* 8192 samples). This means that when the average number of samples in our results is, for example, around 20 000, only 5 iterations have been taken on average, and every iteration adds or removes about 20% of the samples from the test.



(a) c: 1 023, States: 2 096 128, Transitions: 7 328 771



(b) c: 4 095, States: 33 550 336, Transitions: 117 395 459

Fig. 6. Tandem queueing network (CTMC - macOS) [8]

4. In general, the more strata we use, the greater reduction in the number of samples we observe. Also, the performance of antithetic sampling is similar to the case of using only two strata. Our best results are obtained when 4096^1 is used for the number of strata. For example, in Fig. 5a, comparing SPRT and 4096^1 strata shows almost ten times reduction in the average number of samples. The tool's webpage contains an example in which stratification reduces variance to 0. This results in the termination of the algorithm immediately after a minimum number of samples have been taken, giving us 3 orders of magnitude reduction in the number of samples.

5 Conclusion

We presented our new tool called `STMC` for statistical model checking of discrete and continuous Markov chains. It uses antithetic and stratified sampling

to improve the performance of a test. We evaluated our tool on hundreds of examples. Our experimental results show that our techniques can significantly reduce the number of samples and hence, the amount of time required for a test. For example, when 4096^1 strata were used, our algorithms reduced the number of samples more than 3 times on average. We have implemented our tool in `PRISM`, and published it online under GNU General Public License v3.0. We would like to extend `STMC` to support other stratification-based algorithms. In particular, stratified sampling in model checking Markov decision processes, and temporal properties that are defined on the sequence of distributions generated by different types of Markov chains (see [21–23] for examples).

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018)
2. Ballarini, P., Djafri, H., Duflot, M., Haddad, S., Pekergin, N.: COSMOS: a statistical model checker for the hybrid automata stochastic logic. In: 2011 Eighth International Conference on Quantitative Evaluation of SysTems, pp. 143–144 (2011)
3. Barbot, B., Bérard, B., Duplouy, Y., Haddad, S.: Statistical Model-Checking for Autonomous Vehicle Safety Validation. In: SIA Simulation Numérique. Société des Ingénieurs de l'Automobile (2017)
4. Basu, S., Ghosh, A.P., He, R.: Approximate model checking of PCTL involving unbounded path properties. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 326–346. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10373-5_17
5. Bauer, M.S., Mathur, U., Chadha, R., Sistla, A.P., Viswanathan, M.: Exact quantitative probabilistic model checking through rational search. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design, FMCAD 2017, pp. 92–99. FMCAD Inc., Austin (2017)
6. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Runtime verification of biological systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012. LNCS, vol. 7609, pp. 388–404. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_29
7. Fan, J., Zhang, C., Zhang, J.: Generalized likelihood ratio statistics and Wilks phenomenon. Ann. Stat. **29**(1), 153–193 (2001)
8. Hermanns, H., Meyer-Kayser, J., Siegle, M.: Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In: Plateau, B., Stewart, W., Silva, M. (eds.) Proceedings of the 3rd International Workshop on Numerical Solution of Markov Chains (NSMC 1999), pp. 188–207. Prensas Universitarias de Zaragoza (1999)
9. Jegourel, C., Legay, A., Sedwards, S.: A platform for high performance statistical model checking – PLASMA. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 498–503. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_37
10. Katoen, J., Khattri, M., Zapreevt, I.S.: A Markov reward model checker. In: Second International Conference on the Quantitative Evaluation of Systems (QEST 2005), pp. 243–244 (2005)
11. Kwiatkowska, M., Norman, G., Parker, D.: Controller dependability analysis by probabilistic model checking. Control. Eng. Pract. **15**(11), 1427–1434 (2006)

12. 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
13. Lai, T.L.: Sequential Analysis: Some Classical Problems and New Challenges. Statistica Sinica **11**(2), 303–351 (2001)
14. Larsen, K.G., Legay, A.: Statistical model checking: past, present, and future. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 3–15. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_1
15. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_11
16. Muppala, J., Ciardo, G., Trivedi, K.: Stochastic reward nets for reliability prediction. Commun. Reliab. Maint. Serv. **1**(2), 9–20 (1994)
17. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S.: Evaluating the reliability of NAND multiplexing with PRISM. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **24**(10), 1629–1637 (2005)
18. Roohi, N., Wang, Y., West, M., Dullerud, G.E., Viswanathan, M.: Statistical verification of the Toyota powertrain control verification benchmark. In: 20th ACM International Conference on Hybrid Systems: Computation and Control (HSCC), pp. 65–70. ACM (2017)
19. Sen, K., Viswanathan, M., Agha, G.: VESTA: A statistical model-checker and analyzer for probabilistic systems. In: Second International Conference on the Quantitative Evaluation of Systems, pp. 251–252 (2005)
20. Wald, A.: Sequential tests of statistical hypotheses. Ann. Math. Stat. **16**(2), 117–186 (1945)
21. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.: A Mori-Zwanzig and MITL based approach to statistical verification of continuous-time dynamical systems. In: International Federation of Automatic Control (IFAC PapersOnLine), vol. 48, no. 27, pp. 267–273 (2015)
22. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.: Statistical verification of dynamical systems using set oriented methods. In: Hybrid Systems: Computation and Control (HSCC), pp. 169–178 (2015)
23. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.: Verifying continuous-time stochastic hybrid systems via Mori-Zwanzig model reduction. In: 2016 IEEE 55th Conference on Decision and Control (CDC), pp. 3012–3017 (2016)
24. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.E.: Statistical Verification of PCTL Using Antithetic and Stratified Samples. Form. Methods Syst. Des. **54**, 145–163 (2019). https://doi.org/10.1007/s10703-019-00339-8
25. Wang, Y., Roohi, N., West, M., Viswanathan, M., Dullerud, G.E.: Statistical verification of PCTL using stratified samples. In: 6th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS), IFAC-PapersOnLine, vol. 51, pp. 85–90 (2018)
26. Welford, B.P.: Note on a method for calculating corrected sums of squares and products. Technometrics **4**(3), 419–420 (1962)

27. Younes, H.L.S.: Ymer: a statistical model checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_43
28. Younes, H.L.S.: Error control for probabilistic model checking. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 142–156. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_10

AMYTISS: Parallelized Automated Controller Synthesis for Large-Scale Stochastic Systems

Abolfazl Lavaei[1(✉)], Mahmoud Khaled[2], Sadegh Soudjani[3], and Majid Zamani[1,4]

[1] Department of Computer Science, LMU Munich, Munich, Germany
lavaei@lmu.de
[2] Department of Electrical Engineering, TU Munich, Munich, Germany
[3] School of Computing, Newcastle University, Newcastle upon Tyne, UK
[4] Department of Computer Science, University of Colorado Boulder, Boulder, USA

Abstract. In this paper, we propose a software tool, called AMYTISS, implemented in C++/OpenCL, for designing correct-by-construction controllers for large-scale discrete-time stochastic systems. This tool is employed to (i) build finite Markov decision processes (MDPs) as finite abstractions of given original systems, and (ii) synthesize controllers for the constructed finite MDPs satisfying bounded-time high-level properties including safety, reachability and reach-avoid specifications. In AMYTISS, scalable parallel algorithms are designed such that they support the parallel execution within CPUs, GPUs and hardware accelerators (HWAs). Unlike all existing tools for stochastic systems, AMYTISS can utilize high-performance computing (HPC) platforms and cloud-computing services to mitigate the effects of the state-explosion problem, which is always present in analyzing large-scale stochastic systems. We benchmark AMYTISS against the most recent tools in the literature using several physical case studies including robot examples, room temperature and road traffic networks. We also apply our algorithms to a 3-dimensional autonomous vehicle and 7-dimensional nonlinear model of a BMW 320i car by synthesizing an autonomous parking controller.

Keywords: Parallel algorithms · Finite MDPs · Automated controller synthesis · Discrete-time stochastic systems · High performance computing platform

1 Introduction

1.1 Motivations

Large-scale stochastic systems are an important modeling framework to describe many real-life safety-critical systems such as power grids, traffic networks, self-driving cars, and many other applications. For this type of complex systems,

A. Lavaei and M. Khaled—Authors have contributed equally.

This work was supported in part by the H2020 ERC Starting Grant AutoCPS (grant agreement No. 804639).

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 461–474, 2020.
https://doi.org/10.1007/978-3-030-53291-8_24

automating the controller synthesis procedure to achieve high-level specifications, *e.g.,* those expressed as linear temporal logic (LTL) formulae [24], is inherently very challenging mainly due to their computational complexity arising from uncountable sets of states and actions. To mitigate the encountered difficulty, finite abstractions, *i.e.,* systems with finite state sets, are usually employed as replacements of original continuous-space systems in the controller synthesis procedure. More precisely, one can first abstract a given continuous-space system by a simpler one, *e.g.,* a finite Markov decision process (MDP), and then perform analysis and synthesis over the abstract model (using algorithmic techniques from computer science [3]). Finally, the results are carried back to the original system, while providing a guaranteed error bound [5,13–21,23].

Unfortunately, construction of finite MDPs for large-scale complex systems suffers severely from the so-called *curse of dimensionality*: the computational complexity grows exponentially as the number of state variables increases. To alleviate this issue, one promising solution is to employ high-performance computing (HPC) platforms together with cloud-computing services to mitigate the state-explosion problem. In particular, HPC platforms have a large number of processing elements (PEs) and this significantly affects the time complexity when serial algorithms are parallelized [7].

1.2 Contributions

The main contributions and merits of this work are:

(1) We propose a novel data-parallel algorithm for constructing finite MDPs from discrete-time stochastic systems and storing them in efficient distributed data containers. The proposed algorithm handles large-scale systems.
(2) We propose a parallel algorithm for synthesizing discrete controllers using the constructed MDPs to satisfy safety, reachability, or reach-avoid specifications. More specifically, we introduce a parallel algorithm for the iterative computation of Bellman equation in standard dynamic programming [26,27].
(3) Unlike the existing tools in the literature, AMYTISS accepts bounded disturbances and natively supports both additive and multiplicative noises with different practical distributions including normal, uniform, exponential, and beta.

We apply the proposed implementations to real-world applications including robot examples, room temperature and road traffic networks, and autonomous vehicles. This extends the applicability of formal methods to some safety-critical real-world applications with high dimensions. The results show remarkable reductions in the memory usage and computation time outperforming all existing tools in the literature.

We provide AMYTISS as an *open-source* tool. After compilation, AMYTISS is loaded via pFaces [10] and launched for parallel execution

within available parallel computing resources. The source of AMYTISS and detailed instructions on its building and running can be found in: https://github.com/mkhaled87/pFaces-AMYTISS

Due to lack of space, we provide details of traditional serial and proposed parallel algorithms, case studies, etc. in an arXiv version of the paper [12].

1.3 Related Literature

There exist several software tools on verification and synthesis of stochastic systems with different classes of models. SReachTools [30] performs stochastic reachability analysis for linear, potentially time-varying, discrete-time stochastic systems. ProbReach [25] is a tool for verifying the probabilistic reachability for stochastic hybrid systems. SReach [31] solves probabilistic bounded reachability problems for two classes of models: (i) nonlinear hybrid automata with parametric uncertainty, and (ii) probabilistic hybrid automata with additional randomness for both transition probabilities and variable resets. Modest Toolset [6] performs modeling and analysis for hybrid, real-time, distributed and stochastic systems. Two competitions on tools for formal verification and policy synthesis of stochastic models are organized with reports in [1,2].

FAUST^2 [29] generates formal abstractions for continuous-space discrete-time stochastic processes, and performs verification and synthesis for safety and reachability specifications. However, FAUST^2 is originally implemented in MATLAB and suffers from the curse of dimensionality due to its lack of scalability for large-scale models. StocHy [4] provides the quantitative analysis of discrete-time stochastic hybrid systems such that it constructs finite abstractions, and performs verification and synthesis for safety and reachability specifications.

AMYTISS differs from FAUST^2 and StocHy in two main directions. First, AMYTISS implements novel parallel algorithms and data structures targeting HPC platforms to reduce the undesirable effects of the state-explosion problem. Accordingly, it is able to perform parallel execution in different heterogeneous computing platforms including CPUs, GPUs and HWAs. Whereas, FAUST^2 and StocHy can only run serially on one CPU, and consequently, it is limited to small systems. Additionally, AMYTISS can handle the abstraction construction and controller synthesis for two and a half player games (*e.g.,* stochastic systems with bounded disturbances), whereas FAUST^2 and StocHy only handle one and a half player games (*e.g.,* disturbance-free systems).

Unlike all existing tools, AMYTISS offers highly scalable, distributed execution of parallel algorithms utilizing all available processing elements (PEs) in any heterogeneous computing platform. To the best of our knowledge, AMYTISS is the only tool of its kind for continuous-space stochastic systems that is able to utilize all types of compute units (CUs), simultaneously.

We compare AMYTISS with FAUST^2 and StocHy in Table 1 in detail in terms of different technical aspects. Although there have been some efforts in FAUST^2 and StocHy for parallel implementations, these are not compatible with HPC platforms. Specifically, FAUST^2 employs some parallelization techniques using parallel

Table 1. Comparison between AMYTISS, FAUST^2 and StocHy based on native features.

Aspect	**FAUST^2**	**StocHy**	**AMYTISS**
Platform	CPU	CPU	All platforms
Algorithms	Serial on HPC	Serial on HPC	Parallel on HPC
Model	Stochastic control systems: linear, bilinear	Stochastic hybrid systems: linear, bilinear	Stochastic control systems: nonlinear
Specification	Safety, reachability	Safety, reachability	Safety, reachability, reach-avoid
Stochasticity	Additive noise	Additive noise	Additive & multiplicative noises
Distribution	Normal, user-defined	Normal, user-defined	Normal, uniform, exponential, beta, user-defined
Disturbance	Not supported	Not supported	Supported

for-loops and sparse matrices inside Matlab, and StocHy uses Armadillo, a multi-threaded library for scientific computing. However, these tools are not designed for the parallel computation on HPC platforms. Consequently, they can only utilize CPUs and cannot run on GPUs or HWAs. In comparison, AMYTISS is developed in OpenCL, a language specially designed for data-parallel tasks, and supports heterogeneous computing platforms combining CPUs, GPUs and HWAs.

Note that FAUST^2 and StocHy do not natively support reach-avoid specifications in the sense that users can explicitly provide some avoid sets. Implementing this type of properties requires some modifications inside those tools. In addition, we do not make a comparison here with SReachTools since it is mainly for stochastic reachability analysis of linear, potentially time-varying, discrete-time stochastic systems, while AMYTISS is not limited to reachability analysis and can handle nonlinear systems as well.

Note that we also provide a script in the tool repository[1] that converts the MDPs constructed by AMYTISS into PRISM-input-files [11]. In particular, AMYTISS can natively construct finite MDPs from continuous-space stochastic control systems. PRISM can then be employed to perform the controller synthesis for those classes of complex specifications that AMYTISS does not support.

2 Discrete-Time Stochastic Control Systems

We formally introduce discrete-time stochastic control systems (dt-SCS) below.

Definition 1. *A discrete-time stochastic control system (dt-SCS) is a tuple*

$$\Sigma = (X, U, W, \varsigma, f)\,, \tag{1}$$

[1] https://github.com/mkhaled87/pFaces-AMYTISS/blob/master/interface/exportPrismMDP.m.

where,

- $X \subseteq \mathbb{R}^n$ *is a Borel space as the state set and* $(X, \mathcal{B}(X))$ *is its measurable space;*
- $U \subseteq \mathbb{R}^m$ *is a Borel space as the input set;*
- $W \subseteq \mathbb{R}^p$ *is a Borel space as the disturbance set;*
- ς *is a sequence of independent and identically distributed (i.i.d.) random variables from a sample space* Ω *to a measurable set* $\mathcal{V}_\varsigma$

$$\varsigma := \{\varsigma(k) : \Omega \to \mathcal{V}_\varsigma,\ k \in \mathbb{N}\};$$

- $f : X \times U \times W \to X$ *is a measurable function characterizing the state evolution of the system.*

The state evolution of Σ, for a given initial state $x(0) \in X$, an input sequence $\nu(\cdot) : \mathbb{N} \to U$, and a disturbance sequence $w(\cdot) : \mathbb{N} \to W$, is characterized by the difference equations

$$\Sigma : x(k+1) = f(x(k), \nu(k), w(k)) + \Upsilon(k), \qquad k \in \mathbb{N}, \tag{2}$$

where $\Upsilon(k) := \varsigma(k)$ with $\mathcal{V}_\varsigma = \mathbb{R}^n$ for the case of the additive noise, and $\Upsilon(k) := \varsigma(k)x(k)$ with $\mathcal{V}_\varsigma$ equals to the set of diagonal matrices of the dimension n for the case of the multiplicative noise [22]. We keep the notation Σ to indicate both cases and use respectively $\Sigma_{\mathfrak{a}}$ and $\Sigma_{\mathfrak{m}}$ when discussing these cases individually.

We should mention that our parallel algorithms are independent of the noise distribution. For an easier presentation of the contribution, we present our algorithms and case studies based on normal distributions but our tool natively supports other practical distributions including uniform, exponential, and beta. In addition, we provide a subroutine in our software tool so that the user can still employ the parallel algorithms by providing the density function of the desired class of distributions.

Remark 1. Our synthesis is based on a max-min optimization problem for two and a half player games by considering the disturbance and input of the system as players [9]. Particularly, we consider the disturbance affecting the system as an adversary and maximize the probability of satisfaction under the worst-case strategy of a rational adversary. Hence, we minimize the probability of satisfaction with respect to disturbances, and maximize it over control inputs.

One may be interested in analyzing dt-SCSs without disturbances (cf. case studies). In this case, the tuple (1) reduces to $\Sigma = (X, U, \varsigma, f)$, where $f : X \times U \to X$, and the Eq. (2) can be re-written as

$$\Sigma : x(k+1) = f(x(k), \nu(k)) + \Upsilon(k), \qquad k \in \mathbb{N}. \tag{3}$$

Note that input models in this tool paper are given inside configuration text files. Systems are described by stochastic difference equations as (2)–(3), and the user should provide the right-hand-side of equations[2]. In the next section, we formally define MDPs and discuss how to build finite MDPs from given dt-SCSs.

[2] An example of such a configuration file is provided at: https://github.com/mkhaled87/pFaces-AMYTISS/blob/master/examples/ex-toy-safety/toy2d.cfg.

3 Finite Markov Decision Processes (MDPs)

A dt-SCS Σ in (1) is *equivalently* represented by the following MDP [8, Proposition 7.6]:

$$\Sigma = (X, U, W, T_{\mathsf{x}}),$$

where the map $T_{\mathsf{x}} : \mathcal{B}(X) \times X \times U \times W \to [0, 1]$, is a conditional stochastic kernel that assigns to any $x \in X$, $\nu \in U$, and $w \in W$, a probability measure $T_{\mathsf{x}}(\cdot|x, \nu, w)$. The alternative representation as the MDP is utilized in [28] to approximate a dt-SCS Σ with a *finite* MDP $\widehat{\Sigma}$ using an abstraction algorithm. This algorithm first constructs a finite partition of the state set $X = \cup_i \mathsf{X}_i$, the input set $U = \cup_i \mathsf{U}_i$, and the disturbance set $W = \cup_i \mathsf{W}_i$. Then representative points $\bar{x}_i \in \mathsf{X}_i$, $\bar{\nu}_i \in \mathsf{U}_i$, and $\bar{w}_i \in \mathsf{W}_i$ are selected as abstract states, inputs, and disturbances. The transition probability matrix for the finite MDP $\widehat{\Sigma}$ is also computed as

$$\hat{T}_{\mathsf{x}}(x'|x, \nu, w) = T_{\mathsf{x}}(\Xi(x')|x, \nu, w), \quad \forall x, x' \in \hat{X}, \forall \nu \in \hat{U}, \forall w \in \hat{W}, \tag{4}$$

where the map $\Xi : X \to 2^X$ assigns to any $x \in X$, the corresponding partition element it belongs to, *i.e.,* $\Xi(x) = \mathsf{X}_i$ if $x \in \mathsf{X}_i$. Since $\hat{X}$, $\hat{U}$ and $\hat{W}$ are finite sets, $\hat{T}_{\mathsf{x}}$ is a static map. It can be represented with a matrix and we refer to it, from now on, as the transition probability matrix.

For a given logic specification φ and accuracy level ϵ, the discretization parameter δ can be selected a priori such that

$$|\mathbb{P}(\Sigma \vDash \varphi) - \mathbb{P}(\widehat{\Sigma} \vDash \varphi)| \leq \epsilon, \tag{5}$$

where ϵ depends on the horizon of formula φ, the Lipschitz constant of the stochastic kernel, and the *state* discretization parameter δ (cf. [28, Theorem 9]). We refer the interested reader to the arXiv version [12] for more details.

In the next sections, we propose novel parallel algorithms for the construction of finite MDPs and the synthesis of their controllers.

4 Parallel Construction of Finite MDPs

In this section, we propose an approach to efficiently compute the transition probability matrix $\hat{T}_{\mathsf{x}}$ of the finite MDP $\widehat{\Sigma}$, which is essential for any controller synthesis procedure, as we discuss later in Sect. 5.

4.1 Data-Parallel Threads for Computing $\hat{T}_{\mathsf{x}}$

The serial algorithm for computing $\hat{T}_{\mathsf{x}}$ is presented in Algorithm 1 in the arXiv version [12]. Computations of mean $\mu = f(\bar{x}_i, \bar{\nu}_j, \bar{w}_k, 0)$, PDF$(x \,|\, \mu, \Sigma)$, where PDF stands for probability density functions and Σ is a noise covariance matrix, and of $\hat{T}_{\mathsf{x}}$ all do not share data from one inner-loop to another. Hence, this is an embarrassingly data-parallel section of the algorithm. pFaces [10] can be utilized to launch necessary number of parallel threads on the employed hardware configuration (HWC) to improve the computation time of the algorithm. Each thread will eventually compute and store, independently, its corresponding values within $\hat{T}_{\mathsf{x}}$.

4.2 Less Memory for Post States in $\hat{T}_{\mathsf{x}}$

$\hat{T}_{\mathsf{x}}$ is a matrix with the dimension of $(n_x \times n_\nu \times n_w, n_x)$. The number of columns is n_x as we need to compute and store the probability for each reachable partition element $\Xi(x'_l)$, corresponding to the representing post state x'_l. Here, we consider the Gaussian PDFs for the sake of a simpler presentation. For simplicity, we now focus on the computation of tuple $(\bar{x}_i, \bar{\nu}_j, \bar{w}_k)$. In many cases, when the PDF is decaying fast, only partition elements near μ have high probabilities of being reached, starting from $\bar{x}_i$ and applying an input $\bar{\nu}_j$.

We set a cutting probability threshold $\gamma \in [0, 1]$ to control how many partition elements around μ should be stored. For a given mean value μ, a covariance matrix $\mathbf{\Sigma}$ and a cutting probability threshold γ, $x \in X$ is called a PDF cutting point if $\gamma = \text{PDF}(x|\mu, \mathbf{\Sigma})$. Since Gaussian PDFs are symmetric, by repeating this cutting process dimension-wise, we end up with a set of points forming a hyper-rectangle in X, which we call it the cutting region and denote it by $\hat{X}^{\mathbf{\Sigma}}_{\gamma}$. This is visualized in Fig. 1 in the arXiv version [12] for a 2-dimensional system. Any partition element $\Xi(x'_l)$ with x'_l outside the cutting region is considered to have zero probability of being reached. Such approximation allows controlling the sparsity of the columns of $\hat{T}_{\mathsf{x}}$. The closer the value of γ to zero, the more accurate $\hat{T}_{\mathsf{x}}$ in representing transitions of $\widehat{\Sigma}$. On the other hand, the closer the value of γ to one, less post state values need to be stored as columns in $\hat{T}_{\mathsf{x}}$. The number of probabilities to be stored for each $(\bar{x}_i, \bar{\nu}_j, \bar{w}_k)$ is then $|\hat{X}^{\mathbf{\Sigma}}_{\gamma}|$.

Note that since $\mathbf{\Sigma}$ is fixed prior to running the algorithm, number of columns needed for a fixed γ can be identified before launching the computation. We can then accurately allocate a uniform fixed number of memory locations for any tuple $(\bar{x}_i, \bar{\nu}_j, \bar{w}_k)$ in $\hat{T}_{\mathsf{x}}$. Hence, there is no need for a dynamic sparse matrix data structure and $\hat{T}_{\mathsf{x}}$ is now a matrix with a dimension of $(n_x \times n_\nu \times n_w, |\hat{X}^{\mathbf{\Sigma}}_{\gamma}|)$.

4.3 A Parallel Algorithm for Constructing Finite MDP $\widehat{\Sigma}$

We present a novel parallel algorithm (Algorithm 2 in the arXiv version [12]) to efficiently construct and store $\hat{T}_{\mathsf{x}}$ as a successor. We employ the discussed enhancements in Subsect. 4.1 and 4.2 within the proposed algorithm. We do not parallelize the for-loop in Algorithm 2, Step 2, to avoid excessive parallelism (*i.e.,* we parallelize loops only over X and U, but not over W). Note that, practically, for large-scale systems, $|\hat{X} \times \hat{U}|$ can reach up to billions. We are interested in the number of parallel threads that can be scheduled reasonably by available HW computing units.

5 Parallel Synthesis of Controllers

In this section, we employ dynamic programming to synthesize controllers for constructed finite MDPs $\widehat{\Sigma}$ satisfying safety, reachability, and reach-avoid properties [26,27]. The classical serial algorithm and its proposed parallelized version are respectively presented as Algorithms 3 and 4 in the arXiv version [12]. We

should highlight that the parallelism here mainly comes from the parallelization of matrix multiplication and the loop over time-steps cannot be parallelized due to the data dependency. More details can be found in the arXiv version.

5.1 On-the-Fly Construction of $\hat{T}_{\mathsf{x}}$

In AMYTISS, we also use another technique that further reduces the required memory for computing $\hat{T}_{\mathsf{x}}$. We refer to this approach as *on-the-fly abstractions* (OFA). In OFA version of Algorithm 4 [12], we skip computing and storing the MDP $\hat{T}_{\mathsf{x}}$ and the matrix $\hat{T}_{0\mathsf{x}}$ (*i.e.,* Steps 1 and 5). We instead compute the required entries of $\hat{T}_{\mathsf{x}}$ and $\hat{T}_{0\mathsf{x}}$ on-the-fly as they are needed (*i.e.,* Steps 13 and 15). This significantly reduces the required memory for $\hat{T}_{\mathsf{x}}$ and $\hat{T}_{0\mathsf{x}}$ but at the cost of repeated computation of their entries in each time step from 1 to T_d. This gives the user an additional control over the trade-off between the computation time and memory.

5.2 Supporting Multiplicative Noises and Practical Distributions

AMYTISS natively supports multiplicative noises and practical distributions such as uniform, exponential, and beta distributions. The technique introduced in Subsect. 4.2 for reducing the memory usage is also tuned for other distributions based on the support of their PDFs. Since AMYTISS is designed for extensibility, it allows also for customized distributions. Users need to specify their desired PDFs and hyper-rectangles enclosing their supports so that AMYTISS can include them in the parallel computation of $\hat{T}_{\mathsf{x}}$. Further details on specifying customized distributions are provided in the README file.

AMYTISS also supports multiplicative noises as introduced in (2). Currently, the memory reduction technique of Subsect. 4.2 is disabled for systems with multiplicative noises. This means users should expect larger memory requirements for systems with multiplicative noises. However, users can still benefit from the proposed OFA version to compensate for the increase in memory requirement. We plan to include this feature for multiplicative noises in a future update of AMYTISS. Note that for a better demonstration, previous sections were presented by the additive noise and Gaussian normal PDF to introduce the concepts.

6 Benchmarking and Case Studies

AMYTISS is self-contained and requires only a modern C++ compiler. It supports all major operating systems: Windows, Linux and Mac OS. Once compiled, utilizing AMYTISS is a matter of providing text configuration files and launching the tool. AMYTISS implements scalable parallel algorithms that run on top of pFaces [10]. Hence, users can utilize computing power in HPC platforms and cloud computing to scale the computation and control the computational complexities of their problems. Table 2 lists the HW configuration we use to benchmark AMYTISS. The devices range from local devices in desktop computers to advanced compute devices in Amazon AWS cloud computing services.

Table 2. HW configurations for benchmarking AMYTISS.

Id	Description	PEs	Frequency
CPU_1	Local machine: Intel Xeon E5-1620	8	3.6 GHz
CPU_2	Macbook Pro 15: Intel i9-8950HK	12	2.9 GHz
CPU_3	AWS instance `c5.18xlarge`: Intel Xeon Platinum 8000	72	3.6 GHz
GPU_1	Macbook Pro 15 laptop laptop: Intel UHD Graphics 630	23	0.35 GHz
GPU_2	Macbook Pro 15 laptop: AMD Radeon Pro Vega 20	1280	1.2 GHz
GPU_3	AWS p3.2xlarge instance: NVIDIA Tesla V100	5120	0.8 GHz

Table 3 shows the benchmarking results running AMYTISS with these HWCs for several case studies and makes comparisons between AMYTISS, FAUST^2, and StocHy. We employ a machine with Windows operating system (Intel i7@3.6 GHz CPU and 16 GB of RAM) for FAUST^2, and StocHy. It should be mentioned that FAUST^2 predefines a minimum number of representative points based on the desired abstraction error, and accordingly the computation time and memory usage reported in Table 3 are based on the minimum number of representative points. In addition, to have a fair comparison, we run all the case studies with additive noises since neither FAUST^2 nor StocHy supports multiplicative noises.

To show the applicability of our results to large-scale systems, we apply our techniques to several physical case studies. We synthesize controllers for 3- and 5-dimensional *room temperature networks* to keep temperatures in a comfort zone. Furthermore, we synthesize controllers for *road traffic networks* with 3 and 5 dimensions to keep the density of the traffic below some desired level. In addition, we apply our algorithms to a 2-dimensional nonlinear robot and synthesize controllers satisfying safety and reach-avoid specifications. Finally, we consider 3- and 7-dimensional *nonlinear* models of an autonomous vehicle and synthesize reach-avoid controllers to automatically park the vehicles. For details of case studies, see the arXiv version [12].

Table 3 presents a comparison between AMYTISS, FAUST^2 and StocHy w.r.t the computation time and required memory. For each HWC, we show the time in seconds to solve the problem. Clearly, employing HWCs with more PEs reduces the time to solve the problem. This is a strong indication for the scalability of the proposed algorithms. Since AMYTISS is the only tool for stochastic systems that can utilize the reported HWCs, we do not compare it with other similar tools.

In Table 3, first 13 rows, we also include the benchmark provided in StocHy [4, Case study 3]. Table 4 in the arXiv version [12] shows an additional comparison between StocHy and AMYTISS on a machine with the same configuration as the one employed in [4] (a laptop having an Intel Core i7 – 8550U CPU at 1.80GHz with 8 GB of RAM). StocHy suffers significantly from the state-explosion problem as seen from its exponentially growing computation time. AMYTISS, on the other hand, outperforms StocHy and can handle bigger systems using the same hardware.

Table 3. Comparison between AMYTISS, FAUST^2 and StocHy based on their native features for several (physical) case studies. CSB refers to the continuous-space benchmark provided in [4]. † refers to cases when we run AMYTISS with the OFA algorithm. N/M refers to the situation when there is not enough memory to run the case study. N/S refers to the lack of native support for nonlinear systems. (Kx) refers to an 1000-times speedup. The presented speedup is the maximum speedup value across all reported devices. The required memory usage and computation time for FAUST^2 and StocHy are reported for just constructing finite MDPs. The reported times and memories are respectively in seconds and MB, unless other units are denoted.

Problem	Spec.	$\lvert\hat{X} \times \hat{U}\rvert$	T_d	**AMYTISS** (time)							FAUST^2		fStocHy		Speedup w.r.t	
				Mem.	CPU_1	CPU_2	CPU_3	GPU_1	GPU_2	GPU_3	Mem.	Time	Mem.	Time	FAUST	StocHy
2-d StocHy CSB	Safety	4	6	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0001	≤1.0	0.002	8.5	0.015	**20 x**	**150 x**
3-d StocHy CSB	Safety	8	6	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0001	≤1.0	0.002	8.5	0.08	**20 x**	**800 x**
4-d StocHy CSB	Safety	16	6	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0002	≤1.0	0.01	8.5	0.17	**50 x**	**850 Kx**
5-d StocHy CSB	Safety	32	6	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0003	≤1.0	0.01	8.7	0.54	**33 x**	**1.8 Kx**
6-d StocHy CSB	Safety	64	6	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0006	4.251	1.2	9.6	2.17	**2.0 Kx**	**3.6 Kx**
7-d StocHy CSB	Safety	128	6	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0012	38.26	6	12.9	9.57	**5 Kx**	**7.9 Kx**
8-d StocHy CSB	Safety	256	6	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0026	344.3	37	26.6	40.5	**14.2 Kx**	**15.6 Kx**
9-d StocHy CSB	Safety	512	6	1.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0057	3 GB	501	80.7	171.6	**87.8 Kx**	**30.1 Kx**
10-d StocHy CSB	Safety	1024	6	4.0	≤1.0	≤1.0	≤1.0	≤1.0	≤1.0	0.0122	N/M		297.5	385.5	N/A	**32 Kx**
11-d StocHy CSB	Safety	2048	6	16.0	1.0912	≤1.0	≤1.0	≤1.0	≤1.0	0.0284	N/M		1 GB	1708.2	N/A	**60 Kx**
12-d StocHy CSB	Safety	4096	6	64.0	4.3029	4.1969	≤1.0	≤1.0	≤1.0	0.0624	N/M		4 GB	11216	N/A	**179 Kx**
13-d StocHy CSB	Safety	8192	6	256.0	18.681	19.374	1.8515	1.6802	≤1.0	0.1277	N/M		N/A	≥24 h	N/A	**≥676 Kx**
14-d StocHy CSB	Safety	16384	6	1024.0	81.647	94.750	7.9987	7.3489	6.1632	0.2739	N/M		N/A	≥24 h	N/A	**≥320 Kx**
2-d Robot†	Safety	203401	8	≤1.0	8.5299	5.0991	0.7572	≤1.0	≤1.0	0.0154	N/A		N/A		N/A	N/A
2-d Robot	R.Avoid	741321	16	482.16	48.593	18.554	4.5127	2.5311	3.4353	0.3083	N/S		N/S		N/A	N/A
2-d Robot†	R.Avoid	741321	16	4.2484	132.10	41.865	11.745	5.3161	3.6264	0.1301	N/A		N/A		N/A	N/A
3-d Room Temp.	Safety	7776	8	6.4451	0.1072	0.0915	0.0120	≤1.0	≤1.0	0.0018	3.12	1247	N/M		**692 Kx**	N/A

(*continued*)

Table 3. (*continued*)

Problem	Spec.	$\lvert\hat{X} \times \hat{U}\rvert$	T_d	**AMYTISS** (time)							**FAUST**^{2}		**fStocHy**		Speedup w.r.t	
				Mem.	CPU_1	CPU_2	CPU_3	GPU_1	GPU_2	GPU_3	Mem.	Time	Mem.	Time	FAUST	StocHy
3-d Room Temp.†	Safety	7776	8	≤1.0	0.5701	0.3422	0.0627	≤1.0	≤1.0	0.0028	N/A		N/A		N/A	N/A
5-d Room Temp.	Safety	279936	8	3338.4	200.00	107.93	19.376	10.084	N/M	1.8663	2 GB	3248	N/M		**1740 x**	N/A
5-d Room Temp.†	Safety	279936	8	1.36	716.84	358.23	63.758	30.131	22.334	0.5639	N/A		N/A		N/A	N/A
3-d Road Traffic	Safety	2125764	16	1765.7	29.200	131.30	3.0508	5.7345	10.234	1.2895	N/M		N/M		N/A	N/A
3-d Road Traffic†	Safety	2125764	16	14.19	160.45	412.79	13.632	12.707	11.657	0.3062	N/A		N/A		N/A	N/A
5-d Road Traffic	Safety	68841472	7	8797.4	N/M	537.91	38.635	N/M	N/M	4.3935	N/M		N/M		N/A	N/A
5-d Road Traffic†	Safety	68841472	7	393.9	1148.5	1525.1	95.767	44.285	36.487	0.7397	N/A		N/A		N/A	N/A
3-d Vehicle	R.Avoid	1528065	32	1614.7	2.5 h	1.1 h	871.89	898.38	271.41	10.235	N/S		N/S		N/A	N/A
3-d Vehicle†	R.Avoid	1528065	32	11.17	2.8 h	1.9 h	879.78	903.2	613.55	107.68	N/A		N/A		N/A	N/A
7-d BMW 320i	R.Avoid	3937500	32	10169.4	N/M	≥24 h	21.5 h	N/M	N/M	825.62	N/S		N/S		N/A	N/A
7-d BMW 320i†	R.Avoid	3937500	32	30.64	≥24 h	≥24 h	≥24 h	≥24 h	≥24 h	1251.7	N/A		N/A		N/A	N/A

As seen in Table 3, AMYTISS outperforms FAUST^2 and StocHy in all the case studies (maximum speedups up to 692000 times). Moreover, AMYTISS is the only tool that can utilize the available HW resources. The OFA feature in AMYTISS reduces dramatically the required memory, while still solves the problems in a reasonable time. FAUST^2 and StocHy fail to solve many of the problems since they lack the native support for nonlinear systems, they require large amounts of memory, or they do not finish computing within 24 hours.

Note that considering only dimensions of systems can be sometimes misleading. In fact, number of transitions in MDPs ($|\hat{X} \times \hat{U}|$) can give a better judgment on the size of systems since it directly affects the memory/time needed for solving the problem. For instance in Table 3, the number of transitions for the 14-dimensional case study is 16384, while for the 5-dimensional room temperature example is 279936 transitions (*i.e.,* almost 17 times bigger). This means AMYTISS can clearly handle much larger systems than existing tools.

Acknowledgment. The authors would like to thank Thomas Gabler for his help in implementing traditional serial algorithms for the purpose of analysis and then comparing with the parallel ones.

References

1. Abate, A., et al.: ARCH-COMP19 category report: stochastic modelling. EPiC Ser. Comput. **61**, 62–102 (2019)
2. Abate, A., et al.: ARCH-COMP18 category report: Stochastic modelling. In: ARCH@ ADHS, pp. 71–103 (2018)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Cauchi, N., Abate, A.: StocHy: automated verification and synthesis of stochastic processes. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 247–264. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_14
5. Haesaert, S., Soudjani, S.: Robust dynamic programming for temporal logic control of stochastic systems. CoRR abs/1811.11445 (2018). http://arxiv.org/abs/1811.11445
6. 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
7. Jaja, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Boston (1992)
8. Kallenberg, O.: Foundations of Modern Probability. Springer, New York (1997). https://doi.org/10.1007/b98838
9. Kamgarpour, M., Ding, J., Summers, S., Abate, A., Lygeros, J., Tomlin, C.: Discrete time stochastic hybrid dynamical games: Verification & controller synthesis. In: Proceedings of the 50th IEEE Conference on Decision and Control and European Control Conference, pp. 6122–6127 (2011)
10. Khaled, M., Zamani, M.: pFaces: an acceleration ecosystem for symbolic control. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 252–257 (2019)

11. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46029-2_13
12. Lavaei, A., Khaled, M., Soudjani, S., Zamani, M.: AMYTISS: parallelized automated controller synthesis for large-scale stochastic system. arXiv:2005.06191, May 2020
13. Lavaei, A., Soudjani, S., Majumdar, R., Zamani, M.: Compositional abstractions of interconnected discrete-time stochastic control systems. In: Proceedings of the 56th IEEE Conference on Decision and Control, pp. 3551–3556 (2017)
14. Lavaei, A., Soudjani, S., Zamani, M.: Compositional synthesis of finite abstractions for continuous-space stochastic control systems: a small-gain approach. In: Proceedings of the 6th IFAC Conference on Analysis and Design of Hybrid Systems, vol. 51, pp. 265–270 (2018)
15. Lavaei, A., Soudjani, S., Zamani, M.: From dissipativity theory to compositional construction of finite Markov decision processes. In: Proceedings of the 21st ACM International Conference on Hybrid Systems: Computation and Control, pp. 21–30 (2018)
16. Lavaei, A., Soudjani, S., Zamani, M.: Compositional abstraction-based synthesis of general MDPs via approximate probabilistic relations. arXiv: 1906.02930 (2019)
17. Lavaei, A., Soudjani, S., Zamani, M.: Compositional construction of infinite abstractions for networks of stochastic control systems. Automatica **107**, 125–137 (2019)
18. Lavaei, A., Soudjani, S., Zamani, M.: Compositional abstraction-based synthesis for networks of stochastic switched systems. Automatica **114**, 108827 (2020)
19. Lavaei, A., Soudjani, S., Zamani, M.: Compositional abstraction of large-scale stochastic systems: a relaxed dissipativity approach. Nonlinear Anal. Hybrid Syst. **36**, 100880 (2020)
20. Lavaei, A., Soudjani, S., Zamani, M.: Compositional (in)finite abstractions for large-scale interconnected stochastic systems. IEEE Trans. Autom. Control. (2020). https://doi.org/10.1109/TAC.2020.2975812
21. Lavaei, A., Zamani, M.: Compositional construction of finite MDPs for large-scale stochastic switched systems: a dissipativity approach. In: Proceedings of the 15th IFAC Symposium on Large Scale Complex Systems: Theory and Applications 52(3), 31–36 (2019)
22. Li, W., Todorov, E., Skelton, R.E.: Estimation and control of systems with multiplicative noise via linear matrix inequalities. In: Proceedings of the American Control Conference, pp. 1811–1816 (2005)
23. Mallik, K., Schmuck, A., Soudjani, S., Majumdar, R.: Compositional synthesis of finite-state abstractions. IEEE Trans. Autom. Control. **64**(6), 2629–2636 (2019)
24. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
25. Shmarov, F., Zuliani, P.: ProbReach: verified probabilistic delta-reachability for stochastic hybrid systems. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, pp. 134–139 (2015)
26. Soudjani, S.: Formal abstractions for automated verification and synthesis of stochastic systems. Ph.D. thesis, Technische Universiteit Delft, The Netherlands (2014)
27. Soudjani, S., Abate, A.: Adaptive and sequential gridding procedures for the abstraction and verification of stochastic processes. SIAM J. Appl. Dyn. Syst. **12**(2), 921–956 (2013)

28. Soudjani, S., Abate, A., Majumdar, R.: Dynamic Bayesian networks as formal abstractions of structured stochastic processes. In: Proceedings of the 26th International Conference on Concurrency Theory, pp. 1–14 (2015)
29. Soudjani, S.E.Z., Gevaerts, C., Abate, A.: FAUST^2: Formal Abstractions of Uncountable-STate STochastic Processes. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 272–286. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_23
30. Vinod, A.P., Gleason, J.D., Oishi, M.M.: SReachTools: a MATLAB stochastic reachability toolbox. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 33–38 (2019)
31. Wang, Q., Zuliani, P., Kong, S., Gao, S., Clarke, E.M.: SReach: a probabilistic bounded delta-reachability analyzer for stochastic hybrid systems. In: Roux, O., Bourdon, J. (eds.) CMSB 2015. LNCS, vol. 9308, pp. 15–27. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23401-4_3

PRISM-games 3.0: Stochastic Game Verification with Concurrency, Equilibria and Time

Marta Kwiatkowska[1], Gethin Norman[2], David Parker[3](✉), and Gabriel Santos[1]

[1] Department of Computing Science, University of Oxford, Oxford, UK
[2] School of Computing Science, University of Glasgow, Glasgow, UK
[3] School of Computer Science, University of Birmingham, Birmingham, UK
d.a.parker@cs.bham.ac.uk

Abstract. We present a major new release of the PRISM-games model checker, featuring multiple significant advances in its support for verification and strategy synthesis of stochastic games. Firstly, *concurrent* stochastic games bring more realistic modelling of agents interacting in a concurrent fashion. Secondly, *equilibria*-based properties provide a means to analyse games in which competing or collaborating players are driven by distinct objectives. Thirdly, a *real-time* extension of (turn-based) stochastic games facilitates verification and strategy synthesis for systems where timing is a crucial aspect. This paper describes the advances made in the tool's modelling language, property specification language and model checking engines in order to implement this new functionality. We also summarise the performance and scalability of the tool, and describe a selection of case studies, ranging from security protocols to robot coordination, which highlight the benefits of the new features.

1 Introduction

Quantitative verification and strategy synthesis are powerful techniques for the modelling and analysis of computerised systems which require reasoning about *quantitative* aspects such as probability, time or resource usage. They can be used either to produce formal *guarantees* about a system's behaviour, for example relating to its safety, reliability or efficiency, or to synthesise controllers which ensure that such guarantees will be met at runtime. Examples of applications where these techniques have been used include power controllers, unmanned aerial vehicles, autonomous driving and communication protocols.

As computing systems increasingly involve concurrently acting autonomous agents, *game-theoretic* approaches are becoming widespread in computer science as a faithful modelling abstraction. These techniques can be used to reason

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 475–487, 2020.
https://doi.org/10.1007/978-3-030-53291-8_25

about the *competitive* or *collaborative* behaviour of multiple rational agents or entities with distinct goals or objectives. Applications include designing a defence strategy against attackers in a cybersecurity context or building controllers for autonomous robots operating in an unknown or potentially malicious environment. More broadly, game theory techniques such as mechanism design can be used to design protocols that are robust in the context of selfish participants, for example by incorporating *incentive/reward* schemes. They have been successfully deployed in diverse contexts such as network routing [29], auction design [10], public good provisioning [15] and ranking or recommender systems [30].

However, designing game-theoretic systems correctly is a challenge, in view of the complexity of behaviours arising from the interactions between autonomy, concurrency and quantitative rewards. This motivates the development of formal verification techniques to check their correctness and synthesise correct-by-construction strategies for them. Furthermore, many of these applications require reasoning about *stochasticity*: protocols may employ randomisation, e.g., for reliable dissemination across a network, or to minimise the impact of information leakage to an observer; autonomous robots operate in uncertain environments and may use unreliable hardware components or noisy sensors; and data-driven systems such as ranking or navigation systems rely on learnt probabilistic models for their execution.

These challenges have inspired the development of PRISM-games [22], a model checking tool for *stochastic games*. To date, it supports verification and strategy synthesis for *turn-based* stochastic multi-player games (TSGs) using a variety of objectives, expressed in the temporal logic rPATL (probabilistic alternating-time temporal logic with rewards) [8]. This allows specification of *zero-sum* objectives relating to one coalition of players trying to maximise a probabilistic or reward-based objective, while the remaining players form a second coalition trying to minimise the objective. It has also been extended to include (zero-sum) *multi-objective* properties and additional reward measures such as *long-run average* and *ratio reward* [22]. These methods have been successfully applied to several case studies such as autonomous vehicles, user-centric networks, temperature control and an aircraft electric power system [21,23,32].

In this paper, we present PRISM-games 3.0, which significantly extends its predecessor's functionality in several ways [18–20]. First, it supports the modelling and analysis of *concurrent stochastic multi-player games* (CSGs). Previous versions of the tool supported TSGs, in which it is assumed that each state of the game is controlled by a specific player. CSGs allow players to make decisions simultaneously, without knowledge of each other's choices, providing a more realistic model of concurrent execution and decision making. For this, we extend the PRISM-games modelling language, allowing the user to specify concurrency and synchronisation among agents, as well as to associate rewards to either joint or single actions.

In the first instance, PRISM-games now supports verification and strategy synthesis for CSGs using zero-sum specifications in rPATL [19], which we extend to accommodate *instantaneous rewards*. The second major addition to the tool is

the possibility of reasoning about *equilibria-based* properties, which allow players to have distinct, not necessarily conflicting objectives. We extend rPATL to express properties relating to (subgame perfect) *social-welfare optimal Nash equilibria (SWNE)* [20]. This provides synthesis of strategies for all players (or coalitions) from which there is no incentive for any of them to unilaterally deviate in any state of the game, and where the combined probabilities or rewards are maximised (or minimised).

Thirdly, PRISM-games now adds support for *probabilistic timed multi-player games* (TPTGs) [18] (currently just the turn-based variant of the model). These extend stochastic multi-player games with real-valued clocks, in the style of (probabilistic) timed automata. This allows real-time aspects of a system to be more accurately modelled. Using the *digital clocks* approach [18], timed models are automatically translated to discrete-time models in order to be verified.

In this paper, we describe the key enhancements made to the tool, notably to its modelling and property specification languages. We also summarise the results, algorithms and implementation of the verification and strategy synthesis techniques developed [18–20] to support the new functionality. We then describe a selection of case studies which showcase the advantages of the new features, and summarise the performance and scalability of the tool.

PRISM-games is open source and runs on all major operating systems. It is available from the tool's website [34]. Supporting material for the paper, including a virtual machine that allows easy running of the tool and reproduction of the results presented in Sect. 4, can be found at [33].

Related Tools. Other model checking tools have been developed to provide support for games. For non-stochastic games, model checking tools such as PRALINE [5], EAGLE [31] and EVE [16] support Nash equilibria [27], as does MCMAS-SLK [6] via strategy logic. Uppaal Stratego [11] is a tool that uses machine learning, model checking and simulation for the synthesis of strategies for stochastic priced timed games. GAVS+ [9] is a general-purpose tool for algorithmic game solving, supporting TSGs and (non-stochastic) concurrent games, but not CSGs. GIST [7] allows the analysis of ω-regular properties on probabilistic games, but again focuses on turn-based, not concurrent, games. General purpose tools such as Gambit [26] can compute a variety of equilibria but not for stochastic games.

2 Modelling and Property Specification Languages

2.1 Modelling Concurrent and Timed Games

The new features in PRISM-games 3.0 have required some significant enhancements to the language used to specify models. For the addition of real-time aspects (i.e., TPTGs), the changes are a straightforward combination of the existing language features for specifying TSGs in PRISM-games (player specifications and mapping of model states to them) and for probabilistic timed automata in PRISM (clock variables, module invariants, guards and clock resets).

We therefore focus in this paper on the specification of CSGs, where the language changes are more fundamental.

PRISM-games has an existing language for specifying TSGs, which is an extension of the native PRISM modelling language [22]. Components of the system to be modelled are encapsulated as *modules*, whose states are defined by a set of finite-range *variables* and whose behaviour is specified using action-labelled *guarded commands*. In a state, one or more modules can execute a command to make a transition: if the guard (a predicate over state variables) is satisfied, the state can be modified (probabilistically) by applying the *updates* of the command. Multiple modules can execute simultaneously if their commands are labelled with the same action.

```
csg
// Player specification
player p1 mac1 endplayer
player p2 mac2 endplayer
// Max energy per user
const int emax;
// User 1
module mac1
        s1 : [0..1] init 0; // Has user 1 sent?
        e1 : [0..emax] init emax; // Energy level of user 1
        [w1] true -> (s1'=0); // Wait
        [t1] e1>0 -> (s1'=c'?0:1) & (e1'=e1-1); // Transmit
endmodule
// Define second user using module renaming
module mac2 = mac1 [ s1=s2, e1=e2, w1=w2, t1=t2 ] endmodule
```

```
// Probability qi for transmission success when i users send
const double q1;
const double q2;
// Channel (computes joint transmission probabilities)
module channel
        c : bool init false; // Did a collision occur during transmission?
        [t1,w2] true -> q1:(c'=false) + (1-q1):(c'=true); // User 1 transmits
        [w1,t2] true -> q1:(c'=false) + (1-q1):(c'=true); // User 2 transmits
        [t1,t2] true -> q2:(c'=false) + (1-q2):(c'=true); // Both transmit
endmodule
```

```
// Reward structures
rewards "mess1" // Number of messages sent by user 1
        s1=1 : 1;
endrewards
rewards "mess2" // Number of messages sent by user 2
        s2=1 : 1;
endrewards
rewards "send2" // Number of times users 1 and 2 transmit simultaneously
        [t1,t2] true : 1;
endrewards
```

Fig. 1. An example PRISM-games 3.0 CSG model of medium access control.

CSGs cannot naturally be modelled with this approach for several reasons: (i) players need to be able to concurrently choose between multiple commands with different action labels; (ii) the update performed by one player may be different depending on the action chosen by another player; (iii) when multiple

players execute, variables may need to be updated according to an arbitrary probability distribution, rather than being limited to the product of separate distributions specified locally by individual modules.

Figure 1 shows an example of the PRISM-games 3.0 modelling language, which we use to illustrate some of its new features. It models a probabilistic version of the *medium access control* problem, previously described in [5]. Two users share a communication channel. At each time step, user `maci` (`i` = 1, 2) can choose between transmitting a message (`ti`) or waiting (`wi`). Variable `si` tracks whether a user successfully sent its message in the last time step and `ei` represents its energy level: transmissions can only occur when energy is positive. A third component is the channel `channel`, modelled by Boolean variable `c` denoting whether a collision occurred on the last transmission attempt.

The first difference (with respect to modelling of TSGs) is the player specification: players are associated with modules (rather than states). In the example, module `maci` constitutes player `i`. Modules with no nondeterministic choice (like `channel`) do not need to be tied to a player.

In each state of the CSG, each player chooses between enabled commands of the corresponding modules; if no command is enabled, the player idles. The players move simultaneously so transitions are labelled with *lists* of action labels $[\mathtt{a}_1, \ldots, \mathtt{a}_n]$. So the guarded command notation is extended accordingly: note how the channel's behaviour depends on which actions the two users take (the same principle applies when specifying reward structures; see `send2`). Furthermore, variable updates within a command can now be dependent on the updated values of other variables, provided there are no cyclic dependencies. See for example `(s1'=c'?0:1)`, which updates `s1` depending on whether there was a channel collision (reflected in `c'`, the updated value of `c`). We use this mechanism to model interference on the channel: module `channel` specifies a joint probability distribution which is used to update variables `s1` and `s2` simultaneously.

2.2 Property Specification

PRISM-games 3.0 also extends the language used to specify properties for verification and strategy synthesis. The previous version already supported *zero-sum* queries for TSGs using the logic rPATL, which combines the game logic ATL with reward-based extensions of the probabilistic logic PCTL. Again, for the new real-time models, it is relatively easy to combine the existing rPATL notation with real-valued time bounds. So, we focus here on the case of CSGs, and in particular *equilibria-based* properties.

We compute values or synthesise strategies which are *social-welfare optimal Nash equilibria (SWNE)*, i.e., which maximise (or minimise) the sum of the values associated to the objectives for each player, but from which there is no incentive for any of them to unilaterally deviate in any state of the game. We express such properties by adding to rPATL the + operator, which is then used to denote the sum of the values associated to both *bounded* and *unbounded* objectives.

When using the rewards operator in equilibria-based properties, we can reason about *cumulative* ($\mathtt{C}^{\leqslant \mathtt{k}}$), *instantaneous* ($\mathtt{I}^{=\mathtt{k}}$) and *expected reachability* ($\mathtt{F}$)

objectives. For properties with the probability operator, we support bounded and unbounded reachability using the temporal operators *next* (`X`), *eventually* (`F`) and *until* (`U`). In order to express zero-sum properties for CSGs, we have implemented all the previous temporal operators for probabilistic queries and a subset of the rPATL operators reported in [8] for reward-based queries, adding to that the instantaneous reward operator.

Finally, following the style of rPATL we separate players into *coalitions* with the syntax $\langle\langle coalition \rangle\rangle$, in order to specify the player or association of players for which we seek to maximise or minimise the values for a given zero-sum property. For equilibria-based properties, given that we maximise/minimise the sum, we use the same operator to separate players in different coalitions using a colon, while players in the same coalition are separated by a comma.

The following are examples of both zero-sum and equilibria-based properties for the medium access CSG model described in Fig. 1.

- $\langle\langle$`p1`$\rangle\rangle$`P`$_{\max=?}$`[ s2=0 U s1=1 ]` – what is the maximum probability user 1 can ensure of being the first to transmit, regardless of the behaviour of user 2?
- $\langle\langle$`p2`$\rangle\rangle$`R`$^{\texttt{mess2}}_{\geqslant 2.0}$`[ F e2=0 ]` – can user 2 ensure the expected number of messages it sends before running out of energy is at least 2, whatever user does?
- $\langle\langle$`p1:p2`$\rangle\rangle_{\max\geqslant 2}$(`P[ F s1=1 ] + P[ F s2=1 ]`) – if each user's objective is to send their packet with the maximum probability, is it possible for them to collaborate and both transmit their packets with probability 1?
- $\langle\langle$`p1:p2`$\rangle\rangle_{\max=?}$(`P[ s2=0 U s1=1 ] + P[ s1=0 U s2=1 ]`) – what is the sum of SWNE values if each user tries to maximise the probability of being the first to successfully transmit?
- $\langle\langle$`p1:p2`$\rangle\rangle_{\max=?}$(`R`$^{\texttt{mess1}}$`[ F e1=0 ] + R`$^{\texttt{mess2}}$`[ C`$^{\leqslant k}$`]`) – what is the sum of SWNE values if user 1 tries to maximise the expected number of packets before running out of energy and user 2 maximises the expected number of packets in the first k steps?

3 Verification and Strategy Synthesis Algorithms

3.1 Zero-Sum Properties for CSGs

When verifying zero-sum properties of CSGs, PRISM-games makes use of the model checking algorithms described in [19], which were based on the methods formulated in [2,3]. We rely on *value iteration* and classical convergence criteria to approximate/compute the values for all states of the game under study, and on solving a *linear program* to compute a *minimax* strategy at each state. This corresponds to solving a *matrix game*, which represents a *one-shot zero-sum* game for the actions of each player in a state. For unbounded properties, the solutions of the matrix games are used to synthesise an optimal (memoryless and randomised) strategy for each player. Prior to this numerical solution phase, we find and remove the states for which the optimal expected reward values are infinite by using the qualitative algorithms developed in [1].

Our current implementation uses the LPsolve [24] library to solve the matrix games at each state. CSGs are built and stored in a explicit-state fashion using an extension of PRISM's Java-implemented *explicit* (sparse-matrix based) engine.

3.2 Equilibria-Based Properties for CSGs

For equilibria-based properties of CSGs, PRISM-games implements the methods described in [20]. We rely on value iteration and *backwards induction* to approximate/compute values and synthesise strategies that are SWNE. For unbounded properties, we can only compute values that are ε-Nash equilibria, since Nash equilibria are not guaranteed to exist. At each state, we solve a *bimatrix game*, which is a representation of a *one-shot nonzero-sum* game and is a linear complementarity problem. We solve these games via *labelled polytopes*, finding all equilibria values through an SMT-based implementation, for which we use third-party SMT solvers Z3 [12] and Yices [13]. We make use of a precomputation step of finding and removing *dominated strategies* in order to minimise the number of calls to the solver.

Unlike zero-sum properties, the synthesised strategies for bounded and unbounded equilibria-based properties require (finite) memory. This is needed due to the fact that a player's choices may change once their objectives have been satisfied. We synthesise strategies by combining the strategy vectors computed for each bimatrix game and the strategy generated by computing optimal values for the MDP resulting from playing the game after either goal has been met. As we use value iteration to approximate values for infinite-horizon properties, we can only synthesise ε-Nash strategy profiles.

3.3 Turn-Based Probabilistic Timed Games

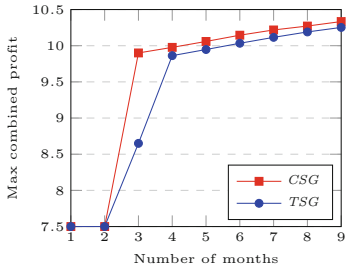
Verification and strategy synthesis of TPTGs relies on the algorithms from [18], which use the *digital clocks* approach that has been a developed for a variety of real-time models. A translation, at the level of the PRISM-games modelling languages, automatically converts the problem of analysing a TPTG into one of solving a (discrete-time) TSG, for which PRISM-games's existing engines can be used. Time-bounded properties are handled by automatically integrating a timing clock into the model prior to translation. As in the rest of PRISM-games, TSGs are also built and solved using the Java-based *explicit* engine.

4 Case Studies and Experimental Results

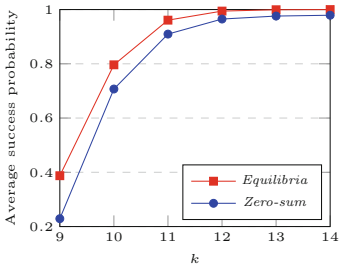
The features added in PRISM-games 3.0 have been used for over 10 new case studies across a wide range of application domains, including computer security (intrusion detection, radio jamming, non-repudiation), communication protocols (medium access control, Aloha), incentive schemes for cooperative networking, multi-robot navigation problems and processor task scheduling. Details can be found in [18–20] and on the case studies section of the PRISM-games website [35].

Supporting material is at [33]. In this section, we showcase four selected case studies that demonstrate the benefits of the tool's new functionality. We also include a discussion of the scalability and performance of the tool.

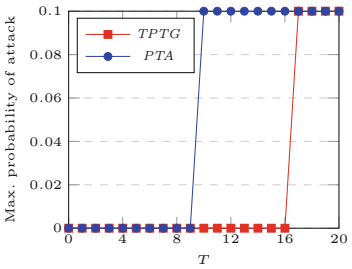
Future Markets Investor. This example models two investors playing against the stock market. Investors choose when to invest or to cash in, and the stock market can decide to bar investments at certain points; fluctuations in share values are modelled stochastically. PRISM-games can, for example, synthesise optimal strategies for the two investors to maximise their expected joint profit over time, acting against the stock market which aims to minimise it.



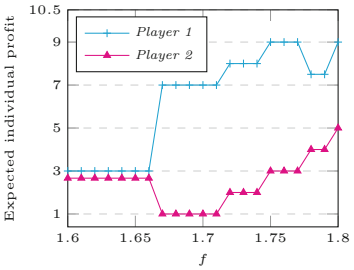
(a) **Future markets investor:** avoiding unrealistic strategy choices using CSGs



(b) **Robot coordination:** using equilibria for mutually beneficial navigation plans



(c) **Non-repudiation:** Attack & defence strategies in a timed, randomised protocol



(d) **Public good game:** Tuning incentive parameter f by synthesising equilibria

Fig. 2. Results illustrating the benefits of the new verification and strategy synthesis techniques implemented in PRISM-games 3.0; see Sect. 4 for details. (Color figure online)

Figure 2(a) shows the results obtained for this property using both a *turn-based* stochastic game (TSG) and a *concurrent* stochastic game (CSG). The former leads to unrealistic modelling as the market can see the choices made by the investors and gain an unfair advantage: the values in the blue plot in Fig. 2(a) are artificially low. In the CSG model, using PRISM 3.0, decisions are taken simultaneously, yielding the correct strategies and values (red plot).

Robot Coordination. Our next example models two robots navigating in opposite directions across a 10-by-10 grid as a CSG. Obstacles which hinder the robots as they move from location to location are modelled stochastically; and if the robots collide, both of them fail in their attempt to reach their goal. We use PRISM-games to find navigation strategies for the two robots, where each robot does not know the choice being made by the other at each step.

The objective for each robot is to navigate successfully, so we maximise the average probability (across the two robots) of success. Figure 2(b) shows the best value that can be achieved within a fixed period of k moves across the grid. One robot aiming single-handedly to achieve this goal performs reasonably well (blue plot), but we can achieve better collective performance by using PRISM-games to synthesise a (social welfare Nash) *equilibrium* strategy (red plot).

Non-repudiation. Next we consider a non-repudiation protocol [25], which permits an originator O to transfer information to a recipient R while guaranteeing non-repudiation, i.e., that neither O nor R can deny that they participated in the transfer. Here, both *probability* (the protocol is randomised) and *time* (the protocol relies on acknowledgement time-outs) are essential ingredients for checking correctness. Furthermore, we model the two participants of the protocol as opposing players, resulting in a TPTG model.

To verify the protocol, we check the worst-case probability that a malicious recipient R can obtain the information being transferred within time T. This can be done with a PTA model (as in [28]) but, with a timed game model, we can also analyse counter-strategies of the honest participant. The results (see Fig. 2(c)) show that, while it is not possible to prevent the information being received, it *is* possible to delay it (the red plot shows lower probabilities for higher times). Note that the bound T is an actual time bound, unlike the examples above, where step-bounded properties measure the number of steps or rounds.

Public Good Game. Lastly, we show a new case study modelling a *public good game*, a well studied model of social choice in economics where participants repeatedly decide how much of an endowment to keep for themselves or to share it with the other players. The total shared by the players is boosted by a factor f in order to incentivise sharing and then divided equally between the players.

Figure 2(d) shows results from a 2-player game, modelled as a CSG. Player choices are necessarily *concurrent*, to avoid cheating. We also need to use *equilibria* since the players have distinct individual goals (maximising personal expected profit). Figure 2(d) shows the values for each player in a synthesised optimal (social welfare Nash) equilibrium for varying f. Changes in f affect both the resulting profit *and* potential inequalities between players in equilibria, indicating the subtleties involved when tuning parameters in an incentive mechanism and the usefulness of analysing this with PRISM-games.

Scalability and Performance. Finally, we show some experimental results for a representative selection of larger examples, to give an indication of the scalability and performance of PRISM-games 3.0. Table 1 shows a range of models (the first 4 are CSGs; the last is a TPTG), the statistics for each one (number of

Table 1. Model statistics for some of the case studies.

Case study	Players	States transitions	Constr. time(s)	Property	Verif. time(s)
Robot coordination	2	159,202 10,765,010	30.94	$\langle\langle p_1 \rangle\rangle \mathtt{P}_{\max=?}[\,\neg \mathtt{c}\, \mathtt{U}^{\leqslant k} \mathtt{g}_1\,]$	114.5
	2	159,202 10,765,010	39.00	$\langle\langle p_1{:}p_2 \rangle\rangle_{\max=?}(\mathtt{P}[\,\neg \mathtt{c}\, \mathtt{U}^{\leqslant k} \mathtt{g}_1\,]{+}\mathtt{P}[\,\neg \mathtt{c}\, \mathtt{U}^{\leqslant k} \mathtt{g}_2\,])$	1,080
Future markets investors	3	1,398,441 7,374,616	51.2	$\langle\langle i_1 \rangle\rangle \mathtt{R}_{\max=?}[\,\mathtt{F}^{c}\ \mathsf{cashed}_1\,]$	1,030
	3	478,761 2,265,560	13.47	$\langle\langle i_1{:}i_2 \rangle\rangle_{\max=?}(\mathtt{R}[\mathtt{F}\ \mathtt{c}_1\,]{+}\mathtt{R}[\mathtt{F}\ \mathtt{c}_2\,])$	13,110
User-centric networks	7	2,993,308 11,392,196	198.6	$\langle\langle \mathit{user} \rangle\rangle \mathtt{R}_{\max=?}[\,\mathtt{F}^{c}\ \mathit{services}{=}K\,]$	1,061
Aloha	3	556,168 2,401,113	15.7	$\langle\langle p_2, p_3 \rangle\rangle \mathtt{R}_{\min=?}[\,\mathtt{F}\ \mathsf{sent}_{2,3}\,]$	317.8
	3	3,334,681 17,834,254	146.1	$\langle\langle p_1{:}p_2{,}p_3 \rangle\rangle_{\min=?}(\mathtt{R}[\mathtt{F}\ \mathtt{s}_1\,]{+}\mathtt{R}[\mathtt{F}\ \mathtt{s}_{2,3}\,])$	3,129
Task graph scheduling	2	659,948 1,798,198	11.16	$\langle\langle \mathit{sched} \rangle\rangle \mathtt{R}_{\max=?}[\,\mathtt{F}\ \mathit{done}\,]$	89.7

players, states, transitions) and the time taken to build and verify the model for some example properties on a 2.10 GHz Intel Xeon with 8 GB of JVM memory.

Verification of CSGs is more computationally expensive than for TSGs supported in earlier versions of the tool, but PRISM-games 3.0 is able to build and analyse CSGs with more than 3 million states on relatively modest hardware. The majority of the time is spent solving (bi)matrix games, which is done repeatedly for all states of the model. Hence, the number of choices per state, which dictates the size of these games, has a greater impact on performance than for TSGs. Unsurprisingly, equilibria properties are slower than zero-sum ones. For both types of property, the number of players in the game does not have a major impact since they are grouped into coalitions yielding a 2-player game to solve. For TPTGs, the digital clocks translation is fast since it is done syntactically, and then a TSG is solved whose size depends on several factors, primarily the number of locations and the magnitude of any time bound in the property.

5 Conclusions

We have presented PRISM-games 3.0, which adds three major new features: (i) concurrent stochastic games; (ii) synthesis of equilibria; and (iii) timed probabilistic games. The usefulness of these has been illustrated on several newly created or extended applications.

CSGs are considerably more expensive to solve than their turn-based counterparts and a key challenge is efficiently solving the matrix game at each state, which is itself a non-trivial optimisation problem. For equilibria, the main difficulty is finding an optimal equilibrium, which currently relies on iteratively restricting the solution search space. Both problems are sensitive to the limitations and issues of floating-point arithmetic, particularly equilibria computation, and might benefit from arbitrary precision representations. Recent research has

also pointed out the shortcomings of only using a lower bound approximation as a stopping criterion for value iteration, as it can lead to inaccuracies [4,14,17]. The impact of similar issues on model checking for games is still to be studied.

A range of further challenges exist for future work. These include providing support for *multi-coalitional* properties and implementing other techniques for equilibria computation. For timed games, we plan to investigate concurrent variants, and also zone-based solution techniques. More broadly speaking, partial information variants of games would be a useful addition.

Acknowledgements. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 834115) and the EPSRC Programme Grant on Mobile Autonomy (EP/M019918/1).

References

1. de Alfaro, L., Henzinger, T.: Concurrent omega-regular games. In: LICS 2000, pp. 141–154 (2000)
2. de Alfaro, L., Henzinger, T., Kupferman, O.: Concurrent reachability games. Theor. Comput. Sci. **386**(3), 188–217 (2007)
3. de Alfaro, L., Majumdar, R.: Quantitative solution of omega-regular games. J. Comput. Syst. Sci. **68**(2), 374–397 (2004)
4. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for Markov decision processes. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 160–180. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_8
5. Brenguier, R.: PRALINE: a tool for computing nash equilibria in concurrent games. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 890–895. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_63
6. Čermák, P., Lomuscio, A., Mogavero, F., Murano, A.: MCMAS-SLK: a model checker for the verification of strategy logic specifications. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 525–532. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_34
7. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Radhakrishna, A.: GIST: a solver for probabilistic games. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 665–669. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_57. pub.ist.ac.at/gist/
8. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Form. Methods Syst. Des. **43**(1), 61–92 (2013)
9. Cheng, C.-H., Knoll, A., Luttenberger, M., Buckl, C.: GAVS+: an open platform for the research of algorithmic game solving. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 258–261. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_22. sourceforge.net/projects/gavsplus/
10. Cramton, P., Shoham, Y., Steinberg, R.: An overview of combinatorial auctions. SIGecom Exch. **7**, 3–14 (2007)

11. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal Stratego. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 206–211. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_16. people.cs.aau.dk/marius/stratego/
12. 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. github.com/Z3Prover/z3
13. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49. yices.csl.sri.com
14. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)
15. Hauser, O., Hilbe, C., Chatterjee, K., Nowak, M.: Social dilemmas among unequals. Nature **572**, 524–527 (2019)
16. Gutierrez, J., Najib, M., Perelli, G., Wooldridge, M.: EVE: a tool for temporal equilibrium analysis. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 551–557. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_35. github.com/eve-mas/eve-parity
17. Kelmendi, E., Krämer, J., Křetínský, J., Weininger, M.: Value iteration for simple stochastic games: stopping criterion and learning algorithm. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 623–642. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_36
18. Kwiatkowska, M., Norman, G., Parker, D.: Verification and control of turn-based probabilistic real-time games. In: Alvim, M.S., Chatzikokolakis, K., Olarte, C., Valencia, F. (eds.) The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy. LNCS, vol. 11760, pp. 379–396. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31175-9_22
19. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Automated verification of concurrent stochastic games. In: McIver, A., Horvath, A. (eds.) QEST 2018. LNCS, vol. 11024, pp. 223–239. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99154-2_14
20. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Equilibria-based probabilistic model checking for concurrent stochastic games. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 298–315. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_19
21. Kwiatkowska, M., Parker, D., Simaitis, A.: Strategic analysis of trust models for user-centric networks. In: Proceedings of the SR'13, EPTCS, vol. 112, pp. 53–60. Open Publishing Association (2013)
22. Kwiatkowska, M., Parker, D., Wiltsche, C.: PRISM-games 2.0: a tool for multi-objective strategy synthesis for stochastic games. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 560–566. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_35
23. Kwiatkowska, M., Parker, D., Wiltsche, C.: PRISM-games: verification and strategy synthesis for stochastic multi-player games with multiple objectives. Softw. Tools Technol. Transf. **20**(2), 195–210 (2018)
24. LPSolve (version 5.5). lpsolve.sourceforge.net/5.5/
25. Markowitch, O., Roggeman, Y.: Probabilistic non-repudiation without trusted third party. In: Proceedings of the 2nd Workshop on Security in Communication Networks (1999)
26. McKelvey, R., McLennan, A., Turocy, T.: Gambit: Software tools for game theory, version 16.0.1 (2016). gambit-project.org

27. Nash, J.: Equilibrium points in n-person games. Proc. Natl. Acad. Sci **36**, 48–49 (1950)
28. Norman, G., Parker, D., Sproston, J.: Model checking for probabilistic timed automata. Form. Methods Syst. Des. **43**(2), 164–190 (2013). https://doi.org/10.1007/s10703-012-0177-x
29. Roughgarden, T., Tardos, E.: How bad is selfish routing? J. ACM **49**, 236–259 (2002)
30. Tennenholtz, M., Kurland, O.: Rethinking search engines and recommendation systems: a game theoretic perspective. Commun. ACM **62**, 66–75 (2019)
31. Toumi, A., Gutierrez, J., Wooldridge, M.: A tool for the automated verification of nash equilibria in concurrent games. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) ICTAC 2015. LNCS, vol. 9399, pp. 583–594. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25150-9_34
32. Wiltsche, C.: Assume-guarantee strategy synthesis for stochastic games. Ph.D. thesis, University of Oxford (2015)
33. Supporting materials and artifact. prismmodelchecker.org/files/cav20pg3/
34. PRISM-games website. prismmodelchecker.org/games/
35. PRISM-games case studies. prismmodelchecker.org/games/casestudies.php

Optimistic Value Iteration

Arnd Hartmanns[1(✉)] and Benjamin Lucien Kaminski[2]

[1] University of Twente,
Enschede, The Netherlands
arnd.hartmanns@utwente.nl
[2] University College London, London, UK
b.kaminski@ucl.ac.uk

Abstract. Markov decision processes are widely used for planning and verification in settings that combine controllable or adversarial choices with probabilistic behaviour. The standard analysis algorithm, value iteration, only provides *lower bounds* on infinite-horizon probabilities and rewards. Two "sound" variations, which also deliver an *upper bound*, have recently appeared. In this paper, we present a new sound approach that leverages value iteration's ability to *usually* deliver good lower bounds: we obtain a lower bound via standard value iteration, use the result to "guess" an upper bound, and prove the latter's correctness. We present this *optimistic value iteration* approach for computing reachability probabilities as well as expected rewards. It is easy to implement and performs well, as we show via an extensive experimental evaluation using our implementation within the mcsta model checker of the Modest Toolset.

1 Introduction

Markov decision processes (MDP, [30]) are a widely-used formalism to represent discrete-state and -time systems in which *probabilistic* effects meet controllable *nondeterministic* decisions. The former may arise from an environment or agent whose behaviour is only known statistically (e.g. message loss in wireless communication or statistical user profiles), or it may be intentional as part of a randomised algorithm (such as exponential backoff in Ethernet). The latter may be under the control of the system—then we are in a planning setting and typically look for a *scheduler* (or strategy, policy) that minimises the probability of unsafe behaviour or maximises a reward—or it may be considered adversarial, which is the standard assumption in verification: we want to establish that the maximum probability of unsafe behaviour is below, or that the minimum reward is above, a specified threshold. Extensions of MDP cover continuous time [11,26],

The authors are listed alphabetically. This work was partly performed while author B. L. Kaminski was at RWTH Aachen University, Aachen, Germany. This work was supported by ERC Advanced Grant 787914 (FRAPPANT), DFG Research Training Group 2236 (UnRAVeL), and NWO VENI grant no. 639.021.754.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 488–511, 2020.
https://doi.org/10.1007/978-3-030-53291-8_26

and the analysis of complex formalisms such as stochastic hybrid automata [13] can be reduced to the analysis of MDP abstractions.

The standard algorithm to compute optimal (maximum or minimum) probabilities or reward values on MDP is *value iteration* (VI). It implicitly computes the corresponding optimal scheduler, too. It keeps track of a value for every state of the MDP, locally improves the values iteratively until a "convergence" criterion is met, and then reports the final value for the initial state as the overall result. The initial values are chosen to be an underapproximation of the true values (e.g. 0 for all states in case of probabilities or non-negative rewards). The final values are then an improved underapproximation of the true values. For unbounded (infinite-horizon) properties, there is unfortunately no (known and practical) convergence criterion that could guarantee a predefined error on the final result. Still, probabilistic model checkers such as PRISM [24] report the final result obtained via simple relative or absolute global error criteria as the definitive probability. This is because, on *most* case studies considered so far, value iteration in fact converges fast enough that the (relative or absolute) difference between the reported and the true value approximately meets the error ϵ specified for the convergence criterion. Only relatively recently has this problem of soundness come to the attention of the probabilistic verification and planning communities [7,14,28]. First highlighted on hand-crafted counterexamples, it has by now been found to affect benchmarks and real-life case studies, too [3].

The first proposal to compute sound reachability probabilities was to use *interval iteration* (II [15], first presented in [14]). The idea is to perform two iterations concurrently, one starting from 0 as before, and one starting from 1. The latter improves an overapproximation of the true values, and the process can be stopped once the (relative or absolute) difference between the two values for the initial state is below the specified ϵ, or at any earlier time with a correspondingly larger but known error. Baier et al. extended interval iteration to expected accumulated reward values [3]; here, the complication is to find initial values that are guaranteed to be an overapproximation. The proposed graph-based (i.e. not numerical) algorithm in practice tends to compute conservative initial values from which many iterations are needed until convergence. More recently, *sound value iteration* (SVI) [31] improved upon interval iteration by computing upper bounds on-the-fly and performing larger value improvements per iteration, for both probabilities and expected rewards. However, we found SVI tricky to implement correctly; some edge cases not considered by the algorithm as presented in [31] initially caused our implementation to deliver incorrect results or diverge on very few benchmarks. Both II and SVI fundamentally depend on the MDP being *contracting*; this must be ensured by appropriate structural transformations, e.g. by collapsing end components, a priori. These transformations additionally complicate implementations, and increase memory requirements.

Our Contribution. We present (in Sect. 4) a new algorithm to compute sound reachability probabilities and expected rewards that is both simple and practically efficient. We first (1) perform standard value iteration until "convergence", resulting in a lower bound on the value for every state. To this we (2) apply specific heuristics to "guess", for every state, a candidate upper bound value.

Further value iterations (3) then confirm (if all values decrease) or disprove (if all values increase, or lower and upper bounds cross) the soundness of the upper bounds. In the latter case, we perform more lower bound iterations with reduced ϵ before retrying from step 2. We combine classic results from domain theory with specific properties of value iteration to show that our algorithm terminates. In problematic cases, many retries may be needed before termination, and performance may be worse than interval or sound value iteration. However, on many existing case studies, value iteration already worked well, and our approach attaches a soundness proof to its result with moderate overhead. We thus refer to it as *optimistic value iteration* (OVI). In contrast to II and SVI, it also works well for non-contracting MDP, albeit without a general termination guarantee. Our experimental evaluation in Sect. 5 uses all applicable models from the Quantitative Verification Benchmark Set [21] to confirm that OVI indeed performs as expected. It uses our publicly available implementations of II, SVI, and now OVI in the mcsta model checker of the Modest Toolset [20].

Related Work. In parallel to [15], the core idea behind II was also presented in [7] (later improved in [2]), embedded in a learning-based framework that manages to alleviate the state space explosion problem in models with a particular structure. In this approach, end components are statistically detected and collapsed on-the-fly. II has recently been extended to stochastic games in [23], offering *deflating* as a new alternative to collapsing end components in MDP. Deflating does not require a structural transformation, but rather extra computation steps in each iteration applied to the states of all (a priori identified) end components.

The only known convergence criterion for pure VI was presented in [9, Sect. 3.5]: if we run VI until the absolute error between two iterations is less than a certain value α, then the computed values at that point are within α of the true values, and can in fact be rounded to the exact true values (as implemented in the *rational search* approach [5]). However, α cannot be freely chosen; it is a fixed number that depends on the size of the MDP and the largest denominator of the (rational) transition probabilities. The number of iterations needed is exponential in the size and the denominators. While not very useful in practice, this establishes an exponential upper bound on the number of iterations needed in unbounded-horizon VI. Additionally, Balaji et al. [4] recently showed the computations in finite-horizon value iteration to be EXPTIME-complete.

As an alternative to the iterative numeric road, guaranteed correct results (modulo implementation errors) can be obtained by using precise rational arithmetic. It does not combine too well with iterative methods like II or SVI due to the increasingly small differences between the values and the actual solution. The probabilistic model checker Storm [10] thus combines topological decomposition, policy iteration, and exact solvers for linear equation systems based on Gaussian elimination when asked to use rational arithmetic [22, Section 7.4.8]. The disadvantage is the significant runtime cost for performing the unlimited-precision calculations, limiting such methods to relatively smaller MDP.

The only experimental evaluations using large sets of benchmarks that we are aware of compared VI with II to study the overhead needed to obtain sound

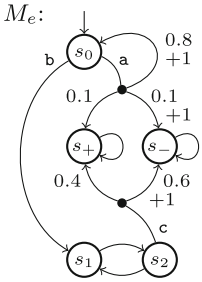


Fig. 1. Example MDP

Table 1. VI and OVI example on M_e

i	$v(s_0)$	$u(s_0)$	$v(s_1)$	$u(s_1)$	$v(s_2)$	$u(s_2)$	*error*	α
0	0		0		0			0.05
1	0.1		0		0.4		0.4	0.05
2	0.18		0.4		0.4		0.4	0.05
3	0.4		0.4		0.4		0.22	0.05
4	0.42	*0.47*	0.4	*0.45*	0.4	*0.45*	0.02	0.05
5	0.436	$\overline{0.47}$	0.4	0.45	0.4	0.45	0.016	
6	0.4488		0.4		0.4		0.0128	0.008
7	0.45904		0.4		0.4		0.01024	0.008
8	0.467232		0.4		0.4		0.008192	0.008
9	0.4737856	*0.5237856*	0.4	*0.45*	0.4	*0.45*	0.0065536	0.008
10	0.47902848	0.51902848	0.4	0.45	0.4	0.45	0.00524288	

results via II [3], and II with SVI to show the performance improvements of SVI [31]. The learning-based method with deflation of [2] does not compete against II and SVI; its aim is rather in dealing with state space explosion (i.e. memory usage). Its performance was evaluated on 16 selected small (<400 k states) benchmark instances in [2], showing absolute errors on the order of 10^{-4} on many benchmarks with a 30-min timeout. SVI thus appears the most competitive technique in runtime and precision so far. Consequently, in our evaluation in Sect. 5, we compare OVI with SVI, and II for reference, using the default relative error of 10^{-6}, including large and excluding clearly acyclic benchmarks (since they are trivial even for VI), with a 10-min timeout which is rarely hit.

2 Preliminaries

$\mathbb{R}_0^+$ is the set of all non-negative real numbers. We write $\{ x_1 \mapsto y_1, \dots \}$ to denote the function that maps all x_i to y_i, and if necessary in the respective context, implicitly maps to 0 all x for which no explicit mapping is specified. Given a set S, its powerset is 2^S. A (discrete) *probability distribution* over S is a function $\mu \in S \to [0,1]$ with countable *support* $spt(\mu) \stackrel{\text{def}}{=} \{ s \in S \mid \mu(s) > 0 \}$ and $\sum_{s \in spt(\mu)} \mu(s) = 1$. $Dist(S)$ is the set of all probability distributions over S.

Markov Decision Processes (MDP) combine nondeterministic choices as in labelled transition systems with discrete probabilistic decisions as in discrete-time Markov chains (DTMC). We define them formally and describe their semantics.

Definition 1. *A* Markov decision process *(MDP) is a triple* $M = \langle S, s_I, T \rangle$ *where* S *is a finite set of* states *with* initial state $s_I \in S$ *and* $T \colon S \to 2^{Dist(\mathbb{R}_0^+ \times S)}$ *is the* transition function. $T(s)$ *must be finite and non-empty for all* $s \in S$.

For $s \in S$, an element of $T(s)$ is a *transition*, and a pair $\langle r, s' \rangle \in spt(T(s))$ is a *branch* to successor state s' with *reward* r and probability $T(s)(\langle r, s' \rangle)$. Let $M^{(s'_I)}$ be M but with initial state s'_I, and M^0 be M with all rewards set to zero.

Example 1. Figure 1 shows our example MDP M_e. We draw transitions as lines to an intermediate node from which branches labelled with probability and reward (if not zero) lead to successor states. We omit the intermediate node and probability 1 for transitions with a single branch, and label some transitions to refer to them in the text. M^e has 5 states, 7 transitions, and 10 branches.

In practice, higher-level modelling languages like MODEST [17] are used to specify MDP. The semantics of an MDP is captured by its *paths*. A path represents a concrete resolution of all nondeterministic and probabilistic choices. Formally:

Definition 2. *A* finite path *is a sequence* $\pi_{\text{fin}} = s_0\, \mu_0\, r_0\, s_1\, \mu_1\, r_1 \ldots \mu_{n-1} r_{n-1} s_n$ *where* $s_i \in S$ *for all* $i \in \{0, \ldots, n\}$ *and* $\exists\, \mu_i \in T(s_i)\colon \langle r_i, s_{i+1}\rangle \in \mathit{spt}(\mu_i)$ *for all* $i \in \{0, \ldots, n-1\}$. *Let* $|\pi_{\text{fin}}| \stackrel{\text{def}}{=} n$, $\text{last}(\pi_{\text{fin}}) \stackrel{\text{def}}{=} s_n$, *and* $\text{rew}(\pi_{\text{fin}}) \stackrel{\text{def}}{=} \sum_{i=0}^{n-1} r_i$. Π_{fin} *is the set of all finite paths starting in* s_I. *A* path *is an analogous infinite sequence* π, *and* Π *is the set of all paths starting in* s_I. *We write* $s \in \pi$ *if* $\exists\, i\colon s = s_i$, *and* $\pi_{\to G}$ *for the shortest prefix of* π *that contains a state in* $G \subseteq S$, *or* $\bot$ *if* π *contains no such state. Let* $\text{rew}(\bot) \stackrel{\text{def}}{=} \infty$.

A scheduler (or *adversary*, *policy* or *strategy*) only resolves the nondeterministic choices of M. For this paper, memoryless deterministic schedulers suffice [6].

Definition 3. *A function* $\mathfrak{s}\colon S \to \mathit{Dist}(\mathbb{R}_0^+ \times S)$ *is a* scheduler *if, for all* $s \in S$, *we have* $\mathfrak{s}(s) \in T(s)$. *The set of all schedulers of* M *is* $\mathfrak{S}(M)$.

Given an MDP M as above, let $M|_{\mathfrak{s}} = \langle S, s_I, T|_{\mathfrak{s}}\rangle$ with $T|_{\mathfrak{s}}(s) = \{\, \mathfrak{s}(s) \,\}$ be the DTMC induced by $\mathfrak{s}$. Via the standard cylinder set construction [12, Sect. 2.2] on $M|_{\mathfrak{s}}$, a scheduler induces a probability measure $\mathbb{P}_{\mathfrak{s}}^M$ on measurable sets of paths starting in s_I. For goal state $g \in S$, the maximum and minimum **probability of reaching** g is defined as $\mathrm{P}_{\max}^M(\diamond\, g) = \sup_{\mathfrak{s}\in\mathfrak{S}} \mathbb{P}_{\mathfrak{s}}^M(\{\, \pi \in \Pi \mid g \in \pi \,\})$ and $\mathrm{P}_{\min}^M(\diamond\, g) = \inf_{\mathfrak{s}\in\mathfrak{S}} \mathbb{P}_{\mathfrak{s}}^M(\{\, \pi \in \Pi \mid g \in \pi \,\})$, respectively. The definition extends to sets G of goal states. Let $R_G^M\colon \Pi \to \mathbb{R}_0^+$ be the random variable defined by $R_G^M(\pi) = \text{rew}(\pi_{\to G})$ and let $\mathbb{E}_{\mathfrak{s}}^M(G)$ be the expected value of R_G^M under $\mathbb{P}_{\mathfrak{s}}^M$. Then the maximum and minimum **expected reward to reach** G is defined as $\mathrm{E}_{\max}^M(G) = \sup_{\mathfrak{s}} \mathbb{E}_{\mathfrak{s}}^M(G)$ and $\mathrm{E}_{\min}^M(G) = \inf_{\mathfrak{s}} \mathbb{E}_{\mathfrak{s}}^M(G)$, respectively. We omit the superscripts for M when they are clear from the context. From now on, whenever we have an MDP with a set of goal states G, we assume that they have been made absorbing, i.e. for all $g \in G$ we only have a self-loop: $T(g) = \{\, \{\, \langle 0, g\rangle \mapsto 1 \,\} \,\}$.

Definition 4. *An* end component *of* M *as above is a (sub-)MDP* $\langle S', T', s_I'\rangle$ *where* $S' \subseteq S$, $T'(s) \subseteq T(s)$ *for all* $s \in S'$, *if* $\mu \in T'(s)$ *for some* $s \in S'$ *and* $\langle r, s'\rangle \in \mathit{spt}(\mu)$ *then* $r = 0$, *and the directed graph with vertex set* S' *and edge set* $\{\, \langle s, s'\rangle \mid \exists\, \mu \in T'(s)\colon \langle 0, s'\rangle \in \mathit{spt}(\mu) \,\}$ *is strongly connected.*

3 Value Iteration

The standard algorithm to compute reachability probabilities and expected rewards is *value iteration* (VI) [30]. In this section, we recall its theoretical foundations and its limitations regarding convergence.

```
1 function GSVI(M = ⟨S, s_I, T⟩, S_?, v, α, diff)
2     repeat
3         error := 0
4         foreach s ∈ S_? do
5             v_new := Φ(v)(s)                          // iterate lower bound
6             if v_new > 0 then error := max(error, diff(v(s), v_new))
7             v(s) := v_new
8     until error ≤ α
```

Algorithm 1. Gauss-Seidel value iteration

3.1 Theoretical Foundations

Let $\mathbb{V} = \{\, v \mid v\colon S \to \mathbb{R}_0^+ \cup \{\infty\} \,\}$ be a space of vectors of values. It can easily be shown that $\langle \mathbb{V}, \preceq \rangle$ with

$$v \preceq w \qquad \text{if and only if} \qquad \forall\, s \in S\colon v(s) \leq w(s)$$

forms a complete lattice, i.e. every subset $V \subseteq \mathbb{V}$ has a supremum (and an infimum) in $\mathbb{V}$ with respect to $\preceq$. We write $v \prec w$ for $v \preceq w \wedge v \neq w$ and $v \not\prec w$ for $\neg(v \preceq w \vee w \preceq v)$.

Minimum and maximum reachability probabilities and expected rewards can be expressed as the *least fixed point* of the *Bellman operator* $\Phi\colon \mathbb{V} \to \mathbb{V}$ given by

$$\Phi(v) \stackrel{\text{def}}{=} \lambda\, s.\ \begin{cases} opt_{\mu \in T(s)} \ \sum_{\langle r, s' \rangle \in spt(\mu)} \ \mu(s') \cdot (r + v(s')) & \text{if } s \in S_? \\ d & \text{if } s \notin S_? \end{cases}$$

where $opt \in \{\, \max, \min \,\}$ and the choice of both $S_? \subseteq S$ and d depends on whether we wish to compute reachability probabilities or expected rewards. In any case, the Bellman operator Φ can be shown to be Scott-continuous [1], i.e. in our case: for any subset $V \subseteq \mathbb{V}$, we have $\Phi(\sup V) = \sup \Phi(V)$.

The Kleene fixed point theorem for Scott-continuous self-maps on complete lattices [1,27] guarantees that $\mathsf{lfp}\,\Phi$, the least fixed point of Φ, indeed exists. Note that Φ can still have more than one fixed point. In addition to mere existence of $\mathsf{lfp}\,\Phi$, the Kleene fixed point theorem states that $\mathsf{lfp}\,\Phi$ can be expressed by

$$\mathsf{lfp}\,\Phi = \lim_{n \to \infty} \Phi^n(\bar{0}) \tag{1}$$

where $\bar{0} \in \mathbb{V}$ is the zero vector and $\Phi^n(v)$ denotes n-fold application of Φ to v. Equation 1 is the basis of VI: the algorithm iteratively constructs a sequence of vectors

$$v_0 = \bar{0} \qquad \text{and} \qquad v_{i+1} = \Phi(v_i),$$

which converges to the sought-after least fixed point. This convergence is *monotonic*: for every $n \in \mathbb{N}$, we have $\Phi^n(\bar{0}) \preceq \Phi^{n+1}(\bar{0})$ and hence $\Phi^n(\bar{0}) \preceq \mathsf{lfp}\,\Phi$. In particular, $\Phi^n(\bar{0})(s_I)$ is an *under*approximation of the sought-after quantity for every n. Note that iterating Φ on *any* underapproximation $v \preceq \mathsf{lfp}\,\Phi$ (instead of $\bar{0}$) will still converge to $\mathsf{lfp}\,\Phi$ and $\Phi^n(v) \preceq \mathsf{lfp}\,\Phi$ will hold for any n.

Gauss-Seidel Value Iteration. Algorithm 1 shows the pseudocode of a VI implementation that uses the so-called *Gauss-Seidel optimisation*: Whereas standard VI needs to store two vectors v_i and v_{i+1}, Gauss-Seidel VI stores only a single vector v and performs updates in place. This does not affect the correctness of VI, but may speed up convergence depending on the order in which the loop in line 4 considers the states in $S_?$. The error metric *diff* is used to check for convergence.

VI for Probabilities. For determining reachability probabilities, we operate on M^0 and set $S_? = S \setminus G$ and $d = 1$. Then the corresponding Bellman operator satisfies

$$(\mathsf{lfp}\, \Phi)(s) = \mathrm{P}_{opt}^{M^{(s)}}(\diamond\, G),$$

and VI will iteratively approximate this quantity *from below*. The corresponding call to Algorithm 1 is $\texttt{GSVI}(M^0,\ S \setminus G,\ \{\, s \mapsto 0 \mid s \in S \setminus G \,\} \cup \{\, s \mapsto 1 \mid s \in G \,\},\ \alpha,\ \mathit{diff})$.

VI for Expected Rewards. For determining the expected reward $\mathrm{E}_{opt}^{M^{(s)}}(G)$, we operate on M and first have to determine the set S_∞ of states from which the minimum (if $opt = \max$) or maximum (if $opt = \min$) probability to reach G is less than 1.[1] If $s_I \in S_\infty$, then the result is ∞ due to the definition of $\mathrm{rew}(\bot)$. Otherwise, we choose $S_? = S \setminus S_\infty$ and $d = \infty$. Then, for $opt = \max$, the least fixed point of the corresponding Bellman operator satisfies

$$(\mathsf{lfp}\, \Phi)(s) = \mathrm{E}_{opt}^{M^{(s)}}(G).$$

Again, VI underapproximates this quantity. The same holds for $opt = \min$ if M does not have end components containing states other than those in G and S_∞. The corresponding call to Algorithm 1 is $\texttt{GSVI}(M,\ S \setminus S_\infty,\ \{\, s \mapsto 0 \mid s \in S \setminus S_\infty \,\} \cup \{\, s \mapsto \infty \mid s \in S_\infty \,\},\ \alpha,\ \mathit{diff})$.

3.2 Uniqueness of Fixed Points

$\mathsf{lfp}\, \Phi$ may not be unique for two reasons: states that cannot reach G under the optimal scheduler may take any value (causing fixed points greater than $\mathsf{lfp}\, \Phi$ for $\mathrm{P}_{\min}$ and $\mathrm{P}_{\max}$), and states in end components may take values higher than $\mathsf{lfp}\, \Phi$. The latter affects $\mathrm{P}_{\max}$ (higher fixed points) and $\mathrm{E}_{\min}$ (lower fixed points).

Example 2. In M_e of Fig. 1, s_1 and s_2 and the two transitions in-between form an end component. For $\mathrm{P}_{\max}^{M_e}(\diamond\, \{\, s_+ \,\})$, $v = \{\, s \mapsto 1 \,\}$ is a non-least fixed point for the corresponding Bellman operator; with appropriate values for s_1 and s_2, we can obtain fixed points with any $v(s_0) > 0.5$ of our choice. Similarly, we have $\mathrm{E}_{\min}^{M}(\{\, s_+, s_- \,\}) = 0.6$ (by scheduling $\mathtt{b}$ in s_0), but due to the end component (with only zero-reward transitions by definition), the fixed point is s.t. $v(s_0) = 0$.

[1] This can be done via Algs. 2 (for $S^1_{\min}$) and 4 (for $S^1_{\max}$) of [12], respectively. These algorithms do not consider the probabilities, but only whether there is a transition and branch (with positive probability) from one state to another or not. We thus call them *graph-based* algorithms, as opposed to *numeric* algorithms like VI itself.

VI works for $\mathrm{P}_{\min}$, $\mathrm{P}_{\max}$, and $\mathrm{E}_{\max}$ with multiple fixed points: we anyway seek $\mathsf{lfp}\,\Phi$ and start from a (trivial) underapproximation. For $\mathrm{E}_{\min}$, (zero-reward) end components need to be collapsed: we determine the maximal end components using algorithms similar to [15, Alg. 1], then replace each of them by a single state, keeping all transitions leading out of the end component. We refer to this as the *ECC* transformation. However, such end components rarely occur in case studies for $\mathrm{E}_{\min}$ since they indicate Zeno behaviour w.r.t. to the reward. As rewards are often associated to time progress, such behaviour would be unrealistic.

To make the fixed points unique, for $\mathrm{E}_{\max}$ and $\mathrm{E}_{\min}$ we fix the values of all states in G to 0. For $\mathrm{P}_{\min}$, we precompute the set $S^0_{\min}$ of states that reach G with minimum probability 0 using Alg. 1 of [12], then fix their values to 0. For $\mathrm{P}_{\max}$, we analogously use $S^0_{\max}$ via Alg. 3 of [12]. For $\mathrm{P}_{\max}$ and $\mathrm{E}_{\min}$, we additionally need to remove end components via ECC. In contrast to the precomputations, ECC changes the structure of the MDP and is thus more memory-intensive.

3.3 Convergence

VI and `GSVI` will not *reach* a fixed point in general, except for special cases such as acyclic MDP. It is thus standard to use a convergence criterion based on the difference between two consecutive iterations (lines 6 and 8) to make `GSVI` terminate: we either check the *absolute error*, i.e.

$$\mathit{diff} = \mathit{diff}_{abs} \stackrel{\text{def}}{=} \lambda\, \langle v_{old}, v_{new} \rangle.\ v_{new} - v_{old},$$

or the *relative error*, i.e.

$$\mathit{diff} = \mathit{diff}_{rel} \stackrel{\text{def}}{=} \lambda\, \langle v_{old}, v_{new} \rangle.\ (v_{new} - v_{old})/v_{new}.$$

By default, probabilistic model checkers like Prism and Storm use diff_{rel} and $\alpha = 10^{-6}$. Upon termination of `GSVI`, v is then closer to the least fixed point, but remains an underapproximation. In particular, α has, in general, no relation to the final difference between $v(s_I)$ and $\mathrm{P}_{opt}(\diamond\, G)$ or $\mathrm{E}_{opt}(G)$, respectively.

Example 3. Consider MDP M_e of Fig. 1 again with $G = \{\, s_+ \,\}$. The first four rows in the body of Table 1 show the values for v after the i-th iteration of the outer loop of a call to $\texttt{GSVI}(M_e^0, \{\, s_0, s_1, s_2 \,\}, \max, \{\, s_+ \mapsto 1 \,\} \cup \{\, s \mapsto 0 \mid s \neq s_+ \,\}, 0.05, \mathit{diff}_{abs})$. After the fourth iteration, `GSVI` terminates since the error is less than $\alpha = 0.05$; at this point, we have $\mathrm{P}_{\max}(\diamond\, s_+) - v(s_0) = 0.08 > \alpha$.

To obtain a value within a prescribed error ϵ of the true value, we can compute an upper bound in addition to the lower bound provided by VI. Interval iteration (II) [3,15] does so by performing, in parallel, a second value iteration on a second vector u that starts from a known overapproximation. For probabilities, the vector $\bar{1} = \{\, s \mapsto 1 \,\}$ is a trivial overapproximation; for rewards, more involved graph-based algorithms need to be used to precompute (a very conservative) one [3]. II terminates when $\mathit{diff}(v(s_I), u(s_I)) \leq 2\epsilon$

Table 2. Preprocessing requirements of value iteration variants

Type	VI	II and SVI	OVI
$\mathrm{P}_{\min}$	–	$S^0_{\min}$	–
$\mathrm{P}_{\max}$	–	$S^0_{\max}$ + ECC	ECC[a]
$\mathrm{E}_{\min}$	$S^1_{\max}$ + ECC	$S^1_{\max}$ + ECC	$S^1_{\max}$ + ECC
$\mathrm{E}_{\max}$	$S^1_{\min}$	$S^1_{\min}$	$S^1_{\min}$

[a]ECC preprocessing for OVI is needed to guarantee termination in theory, however we have not yet found a case study where OVI diverges without ECC.

and returns $v_{II} = \frac{1}{2}(u(s_I) + v(s_I))$. With $v_{true} = \mathrm{P}_{opt}(\Diamond G)$, II thus guarantees that $v_{II} \in [v_{true} - \epsilon \cdot v_{true}, v_{true} + \epsilon \cdot v_{true}]$ and analogously for expected rewards. However, to ensure termination, II requires a unique fixed point: u converges from above to the greatest fixed point $\mathsf{gfp}\,\Phi$, thus for every MDP where $\mathit{diff}((\mathsf{lfp}\,\Phi)(s_I), (\mathsf{gfp}\,\Phi)(s_I)) > 2\epsilon$, II diverges. For $\mathrm{P}_{\max}$, we have $\mathsf{gfp}\,\Phi(s_{ec}) = 1$ for all s_{ec} in end components, thus II tends to diverge when there is an end component. Sound value iteration (SVI) [31] is similar, but uses a different approach to derive upper bounds that makes it perform better overall, and that eliminates the need to precompute an initial overapproximation for expected rewards. However, SVI still requires unique fixed points.

We summarise the preprocessing requirements of VI, II, and SVI in Table 2. With unique fixed points, we can transform $\mathrm{P}_{\min}$ into $\mathrm{P}_{\max}$ by making $S^0_{\min}$ states absorbing and setting G to $S^0_{\min}$, and $\mathrm{P}_{\max}$ into $\mathrm{E}_{\max}$ by a similar transformation adding reward 1 to entering G. Most of the literature on VI variants works in such a setting and describes the $\mathrm{P}_{\max}$ or $\mathrm{E}_{\max}$ case only. Since OVI also works with multiple fixed points, we have to consider all four cases individually.

4 Optimistic Value Iteration

We now present a new, practical solution to the convergence problem for unbounded reachability and expected rewards. It exploits the empirical observation that on many case studies VI delivers results which are roughly α-close to the true value—it only lacks the ability to prove it. Our approach, *optimistic value iteration* (OVI), extends standard VI with the ability to deliver such a proof.

The key idea is to exploit a property of the Bellman operator Φ and its Gauss-Seidel variant as in Algorithm 1 to determine whether a candidate vector is a lower bound, an upper bound, or neither. The foundation is basic domain theory: by Scott-continuity of Φ it follows that Φ is monotonic, meaning $v \preceq w$ implies $\Phi(v) \preceq \Phi(w)$. A principle called *Park induction* [29] for monotonic self-maps on complete lattices yields the following induction rules: For any $u \in \mathbb{V}$,

```
1  function OVI(M = ⟨S, s_I, T⟩, S_?, v, ϵ, α, diff)
2      GSVI(M, S_?, v, α, diff)                          // perform standard value iteration
3      u := { s ↦ diff^+(s) | s ∈ S_? }, viters := 0     // guess candidate upper bound
4      while viters < 1/α do                             // start verification phase
5          up_∀ := true, down_∀ := true, viters := viters + 1, error := 0
6          foreach s ∈ S_? do
7              v_new := Φ(v)(s), u_new := Φ(u)(s)        // iterate both bounds
8              if v_new > 0 then error := max{ error, diff(v(s), v_new) }
9              if u_new < u(s) then                      // upper value decreased:
10                 u(s) := u_new, up_∀ := false          // update u with new lower u_new
11             else if u_new > u(s) then                 // upper value increased:
12                 down_∀ := false                       // discard new higher u_new
13             v(s) := v_new                             // update v with new value v_new
14             if v(s) > u(s) then goto line 17          // lower bound crossed u
15         if down_∀ then return 1/2(u(s_I) + v(s_I))    // u is inductive upper bound
16         else if up_∀ then goto line 17                // u is inductive lower bound
17     return OVI(M, S_?, v, ϵ, error/2, diff)           // retry with reduced α
```

Algorithm 2. Optimistic value iteration

$$\Phi(u) \preceq u \quad \text{implies} \quad \mathsf{lfp}\,\Phi \preceq u. \tag{2}$$

$$\text{and} \quad u \preceq \Phi(u) \quad \text{implies} \quad u \preceq \mathsf{gfp}\,\Phi. \tag{3}$$

Thus, if we can construct a candidate vector u s.t. $\Phi(u) \preceq u$, then u is in fact an upper bound on the sought-after $\mathsf{lfp}\Phi$. We call such a u an *inductive upper bound*. Optimistic value iteration uses this insight and can be summarised as follows:

1. Perform value iteration on v until "convergence" w.r.t. α.
2. Heuristically determine a candidate upper bound u.
3. If $\Phi(u) \preceq u$, then $v \preceq \mathsf{lfp}\,\Phi \preceq u$.
   – If $\mathit{diff}(v(s_I), u(s_I)) \leq 2\epsilon$, terminate and return $\frac{1}{2}\big(u(s_I) + v(s_I)\big)$.
4. If $u \preceq \Phi(u)$ or $u \not\sim v$, then reduce α and go to step 1.
5. Set v to $\Phi(v)$, u to $\Phi(u)$, and go to step 3.

The resulting procedure in more detail is shown as Algorithm 2. Starting from the same initial vectors v as for VI, we first perform standard Gauss-Seidel value iteration (in line 2). We refer to this as the *iteration phase* of OVI. After that, vector v is an improved underapproximation of the actual probabilities or reward values. We then "guess" a vector u of *upper values* from the *lower values* in v (line 3). The guessing heuristics depends on *diff*: if $\mathit{diff} = \mathit{diff}_{abs}$, then we use

$$\mathit{diff}^{+}(s) = \begin{cases} 0 & \text{if } v(s) = 0 \\ v(s) + \epsilon & \text{otherwise;} \end{cases}$$

if $\mathit{diff} = \mathit{diff}_{rel}$, then

$$\mathit{diff}^+(s) = v(s) \cdot (1 + \epsilon).$$

We cap the result at 1 for $\mathrm{P}_{\min}$ and $\mathrm{P}_{\max}$. These heuristics have three important properties: (**H1**) $v(s) = 0$ implies $\mathit{diff}^+(s) = 0$, (**H2**) $\mathit{diff}(v(s), \mathit{diff}^+(s)) \leq 2\epsilon$, and (**H3**) $\mathit{diff}(v(s), \mathit{diff}^+(s)) > 0$ unless $v(s) = 0$ or $v(s) = 1$ for $\mathrm{P}_{\min}$ and $\mathrm{P}_{\max}$.

Then the *verification phase* starts in line 4: we perform value iteration on the lower values v and upper values u at the same time, keeping track of the direction in which the upper values move. For u, line 7 and the conditions around line 10 mean that we actually use operator $\Phi_{\min}(u) = \lambda s.\ \min(\Phi(u)(s), u(s))$. This may shorten the verification phases, and is crucial for our termination argument. A state s is *blocked* if $\Phi(u)(s) > \Phi_{\min}(u)(s)$ and *unblocked* if $\Phi(u)(s) < u(s)$ here.

If, in some iteration, no state was blocked (line 15), then we had $\Phi(u) \preceq u$ before the start of the iteration. We thus know by Eq. 2 that the current u is an inductive upper bound for the values of all states, and the true value must be in the interval $[v(s_I), u(s_I)]$. By property H2, our use of $\Phi_{\min}$ for u, and the monotonicity of Φ as used on v, we also know that $\mathit{diff}(v(s_I), u(s_I)) \leq 2\epsilon$, so we immediately terminate and return the interval's centre $v_I = \frac{1}{2}(u(s_I) + v(s_I))$. The true value $v_{true} = (\mathsf{lfp}\,\Phi)(s_I)$ must then be in $[v_I - \epsilon \cdot v_{true}, v_I + \epsilon \cdot v_{true}]$.

If, in some iteration, no state was unblocked (line 16), then again by Park induction we know that $u \preceq \mathsf{gfp}\,\Phi$. If we are in a situation of unique fixed points, this also means $u \preceq \mathsf{lfp}\,\Phi$, thus the current u is no upper bound: we cancel verification and go back to the iteration phase to further improve v before trying again. We do the same if v crosses u: then $u(s) < v(s) \leq (\mathsf{lfp}\,\Phi)(s)$ for some s, so this u was just another bad guess, too.

Otherwise, we do not yet know the relationship between u and $\mathsf{lfp}\,\Phi$, so we remain in the verification phase until we encounter one of the cases above, or until we exceed the verification budget of $\frac{1}{\alpha}$ iterations (as checked by the loop condition in line 4). This budget is a technical measure to ensure termination.

Optimisation. In case the fixed point of Φ is *unique*, by Park induction (via Eq. 3) we know that $u \preceq \Phi(u)$ implies that u is a lower bound on $\mathsf{lfp}\,\Phi$. In such situations of single fixed points, we can—as an optimisation—additionally replace v by u before the *goto* in line 16.

Heuristics. OVI relies on heuristics to gain an advantage over alternative methods such as II or SVI; it cannot be better on *all* MDP. Concretely, we can choose

1. a stopping criterion for the iteration phase,
2. how to guess candidate upper values from the result of the iteration phase, and
3. how much to reduce α when going back from verification to iteration.

Algorithm 2 shows the choices made by our implementation. We employ the standard stopping criteria used by probabilistic model checkers for VI, and the "weakest" guessing heuristics that satisfies properties H1, H2, and H3 (i.e. guessing any higher values would violate one of these properties). The only arbitrary

choice is how to reduce α, which we at least halve on every retry. We experimentally found this to be a good compromise on benchmarks that we consider in Sect. 5, where

(a) reducing α further causes more and potentially unnecessary iterations in `GSVI` (continuing to iterate when switching to the verification phase would already result in upper values sufficient for termination), and
(b) reducing α less results in more verification phases (whose iterations are computationally more expensive than those of `GSVI`) being started before the values in v are high enough such that we manage to guess a u with $\mathsf{lfp}\,\Phi \preceq u$.

Example 4. We now use the version of Φ to compute $\mathrm{P}_{\max}$ and call

$$\texttt{OVI}(M_e^0, \{ s_0, s_1, s_2 \}, \{ s_+ \mapsto 1 \} \cup \{ s \mapsto 0 \mid s \neq s_+ \}, 0.05, 0.05, \mathit{diff}_{abs}).$$

Table 1 shows the values in v and u during this run, assuming that we use non-Gauss-Seidel iterations. The first iteration phase lasts from $i = 0$ to 4. At this point, u is initialised with the values shown in italics. The first verification phase needs only one iteration to realise that u is actually a lower bound (to a fixed point which is not the least fixed point, due to the uncollapsed end component). Blocked states are marked with a $\overline{\text{bar}}$; unblocked states have a lower u-value than in the previous iteration. We resume `GSVI` from $i = 6$. The error in `GSVI` is again below α, which had been reduced to 0.008, during iteration $i = 9$. We thus start another verification phase, which immediately (in one iteration) finds the newly guessed vector u to be an upper bound, with $\mathit{diff}(v(s_0), u(s_0)) < 2\epsilon$.

4.1 Termination of OVI

We showed above that OVI returns an ϵ-correct result when it terminates. We now show that it terminates in all cases except for $\mathrm{P}_{\max}$ with multiple fixed points. Note that this is a stronger result than what II and SVI can achieve.

Let us first consider the situations where $\mathsf{lfp}\,\Phi$ is the unique fixed point of Φ. First, `GSVI` terminates by Eq. 1. Let us now write v_i and u_i for the vectors u and v as they are at the beginning of verification phase iteration i. We know that $v_0 \preceq u_0$. We distinguish three cases relating the initial guess u_0 to $\mathsf{lfp}\,\Phi$.

1. $u_0 \not\sim \mathsf{lfp}\,\Phi$ or $u_0 \prec \mathsf{lfp}\,\Phi$, i.e. there is a state s with $u_0(s) < (\mathsf{lfp}\,\Phi)(s)$. Since we use $\Phi_{\min}$ on the upper values, it follows $u_i(s) \leq u_0(s) < (\mathsf{lfp}\,\Phi)(s)$ for all i. By Eq. 1, there must thus be a j such that $v_j(s) > u_j(s)$, triggering a retry with reduced α in line 14. Such a retry could also be triggered earlier in line 16. Due to the reduction of α and Eq. 1, every call to `GSVI` will further increase some values in v or reach $v = \mathsf{lfp}\,\Phi$ (in special cases), and for some subsequent guess u we must have $u_0(s) < u(s)$. Consequently, after some repetitions of this case 1, we must eventually guess a u with $\mathsf{lfp}\,\Phi \preceq u$.

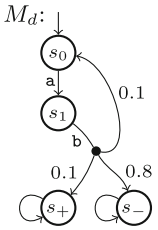


Fig. 2. DTMC M_d

Table 3. Nontermination of OVI on M'_e without ECC

i	$v(s_0)$	$u(s_0)$	$v(s_1)$	$u(s_1)$	$v(s_2)$	$u(s_2)$	*error*	α
0	0		0		0			0.05
1	0.1		0		0.25		0.25	0.05
2	0.18		0.25		0.375		0.25	0.05
3	0.25		0.375		0.4375		0.125	0.05
4	0.375		0.4375		0.46875		0.125	0.05
5	0.4375	*0.5375*	0.46875	*0.56875*	0.484375	*0.584375*	0.0625	0.05
6	0.46875	0.5375	0.484375	0.56875	0.4921875	0.56875	0.03125	
7	0.484375	0.5375	0.4921875	0.56875	0.49609375	0.56875	0.015625	

2. $\mathsf{lfp}\,\Phi \prec u_0$. Observe that operators Φ and $\Phi_{\min}$ are *local* [9], i.e. a state's value can only change if a direct successor's value changes. In particular, a state's value can only decrease (increase) if a direct successor's value decreases (increases). If $u_i(s) < u_{i-1}(s)$, then s cannot be blocked again in any later iteration $j > i$: for it to become blocked, a successor's upper value would have to increase, but $\Phi_{\min}$ ensures non-increasing upper values for all states. Analogously to Eq. 1, we know that [3, Lemma 3.3 (c)]

$$\mathsf{lfp}\,\Phi \preceq u \quad \text{implies} \quad \lim_{n\to\infty} \Phi^n_{\min}(u) = \mathsf{lfp}\,\Phi$$

(for the unique fixpoint case, since [3] assumes contracting MDP as usual). Thus, for all states s, there must be an i such that $u_i(s) < u_{i-1}(s)$; in consequence, there is also an iteration j where no state is blocked any more. Then the condition in line 15 will be true and OVI terminates.
3. $\mathsf{lfp}\,\Phi \preceq u_0$ but not $\mathsf{lfp}\,\Phi \prec u_0$, i.e. there is a state s with $u_0(s) = (\mathsf{lfp}\,\Phi)(s)$. If there is an i where no state, including s, is blocked, then OVI terminates as above. For $\mathrm{P}_{\min}$ and $\mathrm{P}_{\max}$, if $u_0(s) = 1$, s cannot be blocked, so we can w.l.o.g. exclude such s. For other s not to be blocked in iteration i, we must have $u_i(s') = (\mathsf{lfp}\,\Phi)(s')$ for all states s' reachable from s under the optimal scheduler, i.e. all of those states must *reach* the fixed point. This cannot be guaranteed on general MDP. Since this case is a very particular situation unlikely to be encountered in practice with our heuristics, OVI adopts a pragmatic solution: it bounds the number of iterations in every verification phase (cf. line 4). Due to property H3 of our heuristics, $u_0(s) = (\mathsf{lfp}\,\Phi)(s)$ requires $v_0(s) < (\mathsf{lfp}\,\Phi)(s)$, thus some subsequent guess u will have $u(s) > u_0(s)$, and eventually we must get a u with $\mathsf{lfp}\,\Phi \prec u$, which is case 2. Since we strictly increase the iteration bound on every retry, we will eventually encounter case 2 with a sufficiently high bound for termination.

Three of the four situations with multiple fixed points reduce to the corresponding unique fixed point situation due to property H1 of our guessing heuristics:

1. For $\mathrm{P}_{\min}$, recall from Sect. 3.2 that the fixed point is unique if we fix the values of all $S^0_{\min}$ states to 0. In OVI without preprocessing, such states are

in $S_?$, thus they initially have value 0. Φ will not increase their values, neither will guessing due to H1, and neither will $\Phi_{\min}$. Thus OVI here operates on a sublattice of $\langle \mathbb{V}, \preceq \rangle$ where the fixed point of Φ is unique.
2. For $\mathrm{E}_{\min}$, after the preprocessing steps of Table 2, we only need to fix the values of all goal states to 0. Then the argument is the same as for $\mathrm{P}_{\min}$.
3. For $\mathrm{E}_{\max}$, we reduce to a unique fixed point sublattice in the same way, too.

The only case where OVI may not terminate is for $\mathrm{P}_{\max}$ without ECC. Here, end components may cause states to be permanently blocked. However, we did not encounter this on any benchmark used in Sect. 5, so in contrast to e.g. II, OVI is still *practically* useful in this case despite the lack of a termination guarantee.

Example 5. We turn M_e of Fig. 1 into M'_e by replacing the c-labelled transition from s_2 by transition $\{ \langle 0, s_2 \rangle \mapsto \frac{1}{2}, \langle 0, s_+ \rangle \mapsto \frac{1}{4}, \langle 1, s_- \rangle \mapsto \frac{1}{4} \}$, i.e. we can now go from s_2 back to s_2 with probability $\frac{1}{2}$ and to each of s_+, s_- with probability $\frac{1}{4}$. The probability-1 transition from s_2 to s_1 remains. Then Table 3 shows a run of OVI for $\mathrm{P}_{\max}$ with diff_{abs} and $\alpha = 0.1$. s_0 is forever blocked from iteration 6 on.

4.2 Variants of OVI

While the core idea of OVI rests on classic results from domain theory, Algorithm 2 includes several particular choices that work together to achieve good performance and ensure termination. We sketch two variants to motivate these choices.

First, let us use Φ instead of $\Phi_{\min}$ for the upper values, i.e. move the assignment $u(s) := u_{new}$ down into line 13. Then we cannot prove termination because the arguments of case 2 for $\mathsf{lfp}\, \Phi \prec u_0$ no longer hold. Consider DTMC M_d of Fig. 2 and $\mathrm{P}_{\max}(\diamond\, s_+) = \mathrm{P}_{\min}(\diamond\, s_+)$. Let

$$u = \{ s_0 \mapsto 0.2, s_1 \mapsto 1, s_+ \mapsto 1, s_- \mapsto 0 \} \succ \{ s_0 \mapsto \tfrac{1}{9}, s_1 \mapsto \tfrac{1}{9}, \dots \} = \mathsf{lfp}\, \Phi.$$

Iterating Φ, we then get the following sequence of pairs $\langle u(s_0), u(s_1) \rangle$:

$$\langle 0.2, 1 \rangle, \langle 1, 0.12 \rangle, \langle 0.12, 0.2 \rangle, \langle 0.2, 0.112 \rangle, \langle 0.112, 0.12 \rangle, \langle 0.12, 0.1112 \rangle, \dots$$

Observe how the value of s_0 increases iff s_1 decreases and vice-versa. Thus we never encounter an inductive upper or lower bound. In Algorithm 2, we use Gauss-Seidel VI, which would not show the same effect on this model; however, if we insert another state between s_0 and s_1 that is updated last, Algorithm 2 would behave in the same alternating way. This particular u is contrived, but we could have guessed one with a similar relationship of the values leading to similar behaviour.

An alternative that allows us to use Φ instead of $\Phi_{\min}$ is to change the conditions that lead to retrying and termination: We separately store the initial guess of a verification phase as u_0, and then compare each newly calculated u with u_0. If $u \preceq u_0$, then we know that there is an i such that $u = \Phi^i(u) \preceq u_0$.

Φ^i retains all properties of Φ needed for Park induction, so this would also be a proof of $\mathsf{lfp}\ \Phi \preceq u$. The other conditions and the termination proofs can be adapted analogously. However, this variant needs ≈50 % more memory (to store an additional vector of values), and we found it to be significantly slower than Algorthm 2 and the first variant on almost all benchmark instances of Sect. 5.

5 Experimental Evaluation

We have implemented interval iteration (II) (using the "variant 2" approach of [3] to compute initial overapproximations for expected rewards), sound value iteration (SVI), and now optimistic value iteration (OVI) precisely as described in the previous section, in the mcsta model checker of the Modest Toolset [20], which is publicly available at modestchecker.net. It is cross-platform, implemented in C#, and built around the Modest [17] high-level modelling language. Via support for the Jani format [8], mcsta can exchange models with other tools like Epmc [18] and Storm [10]. Its performance is competitive with Storm and Prism [16]. We tried to spend equal effort performance-tuning our VI, II, SVI, and OVI implementations to avoid unfairly comparing highly-optimised OVI code with naïve implementations of the competing algorithms.

In the following, we report on our experimental evaluation of OVI using mcsta on all applicable models of the Quantitative Verification Benchmark Set (QVBS) [21]. All models in the QVBS are available in Jani and can thus be used by mcsta. Most are parameterised, and come with multiple properties of different types. Aside from MDP models, the QVBS also includes DTMCs (which are a special case of MDP), continuous-time Markov chains (CTMC, for which the analysis of unbounded properties reduces to checking the embedded DTMC), Markov automata (MA [11], on which the embedded MDP suffices for unbounded properties), and probabilistic timed automata (PTA [26], some of which can be converted into MDP via the digital clocks semantics [25]). We use all of these model types. The QVBS thus gives rise to a large number of benchmark *instances*: combinations of a model, a parameter valuation, and a property to check. For every model, we chose one instance per probabilistic reachability and expected-reward property such that state space exploration did not run out of memory and VI took at least 10 s where possible. We only excluded

- 2 models with multiple initial states (which mcsta does not yet support),
- 4 PTA with open clock constraints (they cannot be converted to MDP),
- 29 probabilistic reachability properties for which the result is 0 or 1 (they are easily solved by the graph-based precomputations and do not challenge VI),
- 16 instances for which VI very quickly *reaches* the fixed point, which indicates that (the relevant part of) the MDP is acyclic and thus trivial to solve,
- 3 models for which no parameter valuation allowed state space exploration to complete without running out of memory or taking more than 600 s,
- 7 instances where, on the largest state space we could explore, no iterative algorithm took more than 1 s (which does not allow reliable comparisons), and
- the *oscillators* model due to its very large model files,

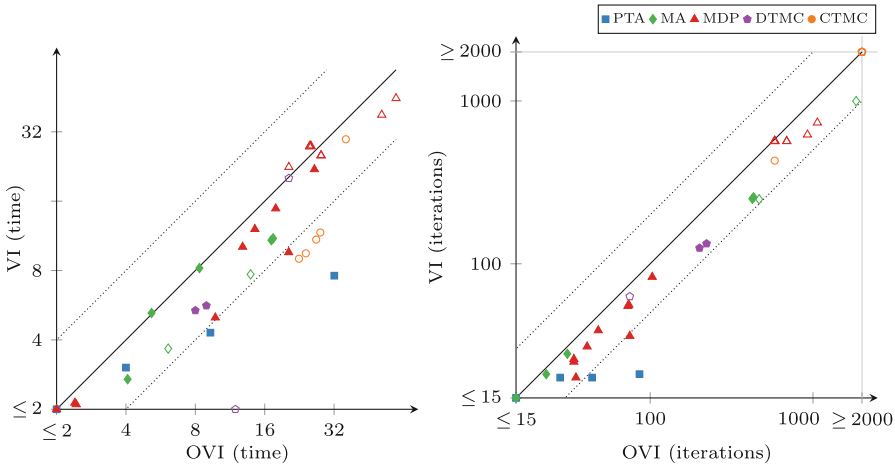


Fig. 3. OVI runtime and iteration count compared to VI (probabilistic reachability)

As a result, we considered 38 instances with probabilistic reachability and 41 instances with expected-reward properties, many comprising several million states.

We ran all experiments on an Intel Core i7-4790 workstation (3.6–4.0 GHz) with 8 GB of memory and 64-bit Ubuntu Linux 18.04. By default, we request a relative half-width of $\epsilon = 10^{-6}$ for the result probability or reward value, and configure OVI to use the relative-error criterion with $\alpha = 10^{-6}$ in the iteration phase. We use a 600 s timeout ("TO"). Due to the number of instances, we show most results as scatter plots like in Fig. 3. Each such plot compares two methods in terms of runtime or number of iterations. Every point $\langle x, y \rangle$ corresponds to an instance and indicates that the method noted on the x-axis took x seconds or iterations to solve this instance while the method noted on the y-axis took y seconds or iterations. Thus points above the solid diagonal line correspond to instances where the x-axis method was faster (or needed fewer iterations); points above (below) the upper (lower) dotted diagonal line are where the x-axis method took less than half (more than twice) as long or as many iterations.

5.1 Comparison with VI

All methods except VI delivered correct results up to ϵ. VI offers low runtime at the cost of occasional incorrect results, and in general the absence of any guarantee about the result. We thus compare with VI separately to judge the overhead caused by performing additional verification, and possibly iteration, phases. This is similar to the comparison done for II in [3]. Figures 3 and 4 show the results. The unfilled shapes indicate instances where VI produced an incorrect result. In terms of runtime, we see that OVI does not often take more than twice as long as VI, and frequently requires less than 50% extra time. On several instances where OVI incurs most overhead, VI produces an incorrect result, indicating

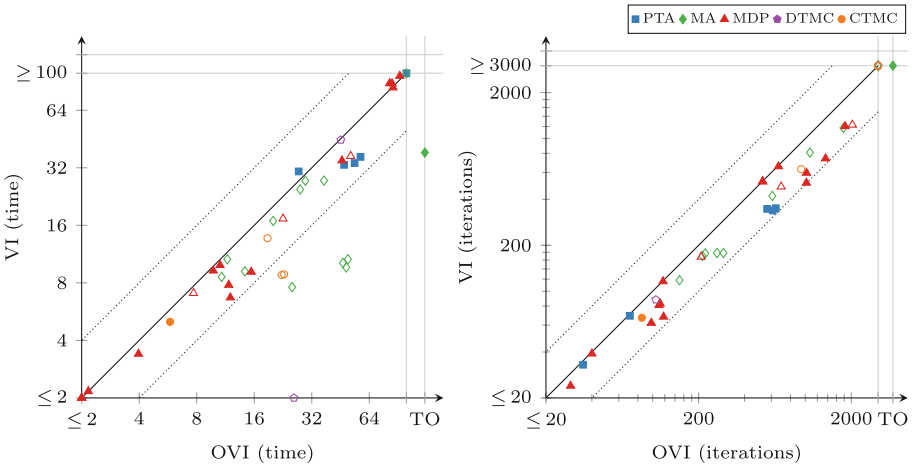


Fig. 4. OVI runtime and iteration count compared to VI (expected rewards)

that they are "hard" instances for value iteration. The unfilled CTMCs where OVI takes much longer to compute probabilities are all instances of the *embedded* model; the DTMC on the x-axis is *haddad-monmege*, an adversarial model built to highlight the convergence problem of VI in [14]. The problematic cases for expected rewards include most MA instances, the two expected-reward instances of the *embedded* CTMC, and again *haddad-monmege*. In terms of iterations, the overhead of OVI is even less than in runtime.

5.2 Comparison with II and SVI

We compare the runtime of OVI with the runtime of II and that of SVI separately for reachability probabilities (shown in Fig. 5) and expected rewards (shown in Fig. 6). As shown in Table 2, OVI has almost the same requirements on precomputations as VI, while II and SVI require extra precomputations and ECC for reachability probabilities. The precomputations and ECC need extra runtime (which turned out to be negligible in some cases but significant enough to cause a timeout in others) prior to the numeric iterations. However, doing the precomputations can reduce the size of the set $S_?$, and ECC can reduce the size of the MDP itself. Both can thus reduce the runtime needed for the numeric iterations. For the overall runtime, we found that none of these effects dominates the other over all models. Thus sometimes it may be better to perform only the required precomputations and transformations, while on other models performing all applicable ones may lead to lower total runtime. For reachability probabilities, we thus compare OVI, II, and SVI in two scenarios: once in the default ("std") setting of mcsta that uses only required preprocessing steps

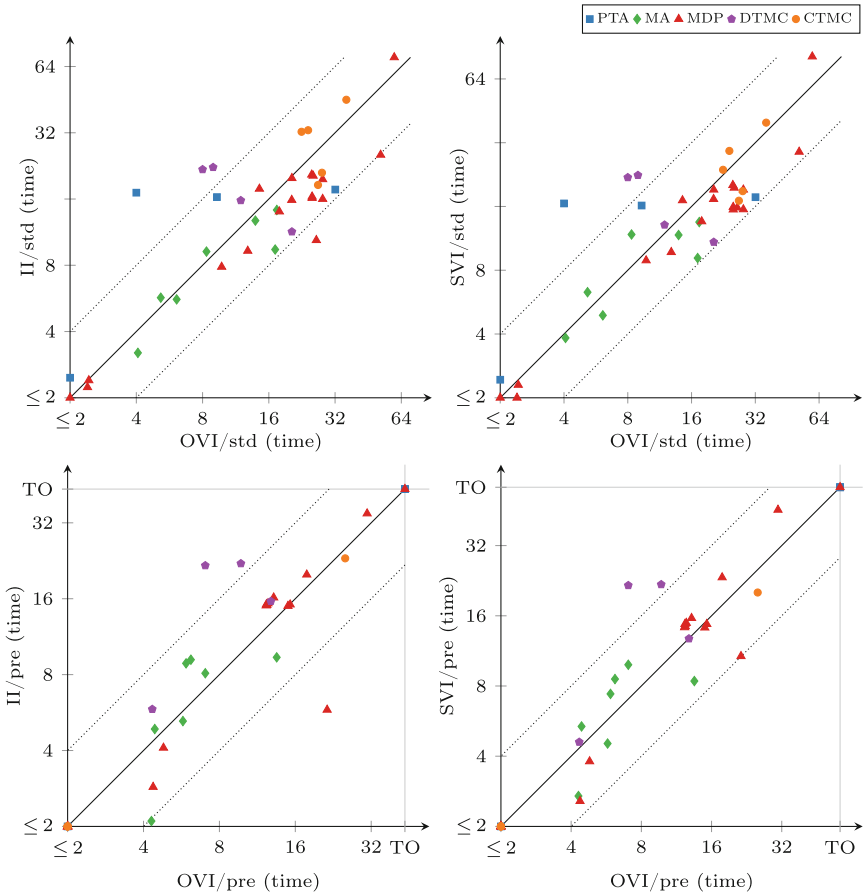


Fig. 5. OVI runtime compared to II and SVI (probabilities)

(without ECC for OVI; we report the total runtime for preprocessing and iterations), and once with all of them enabled ("pre", where we report only the runtime for numeric iterations, plus the computation of initial upper bounds in case of II).

For probabilistic reachability, we see in Fig. 5 that there is no clear winner among the three methods in the "std" setting (top plots). In some cases, the extra precomputations take long enough to give an advantage to OVI, while in others they speed up II and SVI significantly, compensating for their overhead. The "pre" setting (bottom), in which all three algorithms operate on exactly the same input w.r.t. to MDP M and set $S_?$, however, shows a clearer picture: now OVI is faster, sometimes significantly so, than II and SVI on most instances.

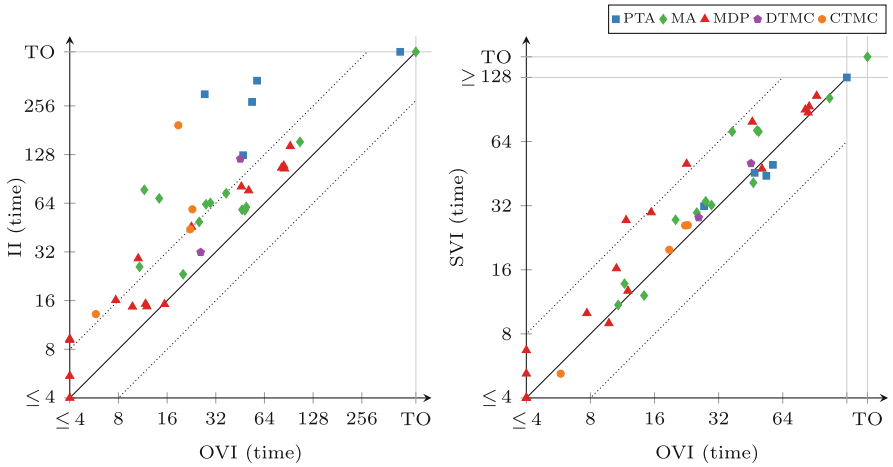


Fig. 6. OVI runtime compared to II and SVI (expected rewards)

Expected-reward properties were more challenging for all three methods (as well as for VI, which produced more errors here than for probabilities). The plots in Fig. 6 paint a very clear picture of OVI being significantly faster for expected rewards than II (which suffers from the need to precompute initial upper bounds that then turn out to be rather conservative), and faster (though by a lesser margin and with few exceptions) than SVI.

In Fig. 7, we give a summary view combining the data from Figs. 3 to 6. For each algorithm, we plot the instances sorted by runtime, i.e. a point $\langle x, y \rangle$ on the line for algorithm z means that some instance took y seconds to solve via z, and there are x instances that z solves in less time. Note in particular that the times are *not* cumulative. The right-hand plot zooms into the left-hand one. We clearly see the speedup offered by OVI over SVI and especially II. Where the scatter plots merely show that OVI often does not obtain more than a 2× speedup compared to SVI, these plots provide an explanation: the VI line is a rough

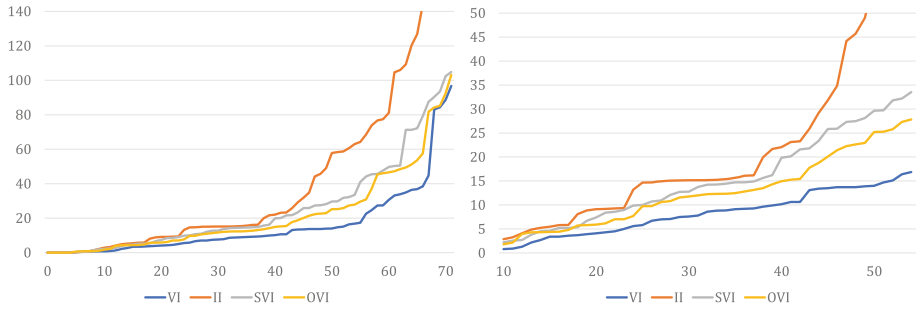


Fig. 7. Summary comparison to VI, II, and SVI, instances ordered by runtime

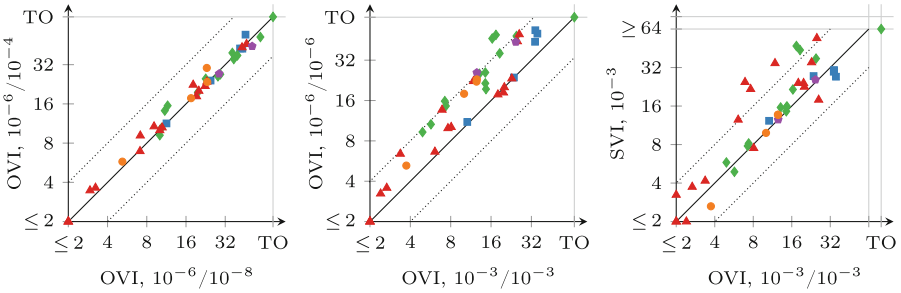


Fig. 8. Influence of ϵ/α on runtime (expected rewards, relative error)

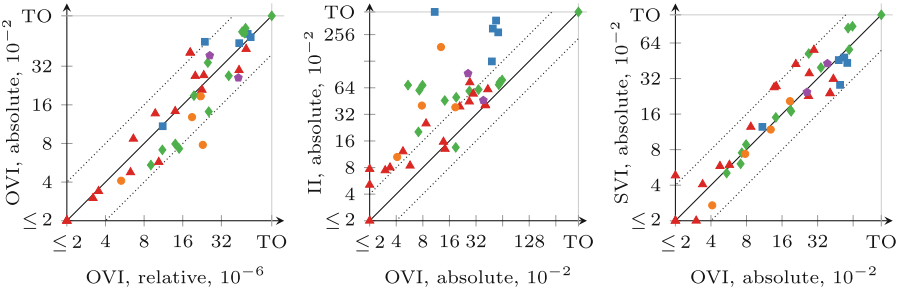


Fig. 9. Runtime comparison with absolute error (expected rewards)

bound on the performance that any *extension* of VI can deliver. Comparing the SVI and VI lines, over much of the plot's range, OVI thus cannot take less than half the runtime of SVI without outperforming VI itself.

5.3 On the Effect of ϵ and α

We also compared the four algorithms for different values of ϵ and, where applicable, α. We show a selection of the results in Fig. 8. The axis labels are of the form "algorithm, ϵ/α". On the left, we see that the runtime of OVI changes if we set α to values different from ϵ, however there is no clear trend: some instances are checked faster, some slower. We obtained similar plots for other combinations of α values, with only a slight tendency towards longer runtimes as $\alpha > \epsilon$. mcsta thus uses $\alpha = \epsilon$ as a default that can be changed by the user.

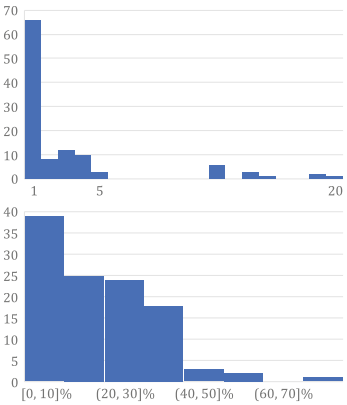
In the middle, we study the impact of reducing the desired precision by setting ϵ to 10^{-3}. This allows OVI to speed up by factors mostly between 1 and 2; the same comparison for SVI and II resulted in similar plots, however VI was able to more consistently achieve higher speedups. When we compare the right plot with the right-hand plot of Fig. 6, we consequently see that the overall result of our comparison between OVI and SVI does not change significantly with the lower precision, although OVI does gain slightly more than SVI.

5.4 Comparing Relative and Absolute Error

In Fig. 9, we show comparison plots for the runtime when using $diff_{abs}$ instead of $diff_{rel}$. Requiring absolute-error-correct results may make instances with low result values much easier and instances with high results much harder. We chose $\epsilon = 10^{-2}$ as a compromise, and the leftmost plot confirms that we indeed chose an ϵ that keeps the expected-reward benchmarks on average roughly as hard as with 10^{-6} relative error. In the middle and right plots, we again see OVI compared with II and SVI. Compared to Fig. 6, both II and SVI gain a little, but there are no significant differences overall. Our experiments thus confirm that the relative performance of OVI is stable under varying precision requirements.

5.5 Verification Phases

On the right, we show histograms of the number of verification phases started (top, from 1 phase on the left to 20 on the right) and the percentage of iterations that are done in verification phases (bottom) over all benchmark instances (probabilities and rewards). We see that, in the vast majority of cases, we need few verification attempts, with many succeeding in the first attempt, and most iterations are performed in the iteration phases.



6 Conclusion

We have presented *optimistic value iteration* (OVI), a new approach to making non-exact probabilistic model checking via iterative numeric algorithms sound in the sense of delivering results within a prescribed interval around the true value (modulo floating-point and implementation errors). Compared to interval (II) and sound value iteration (SVI), OVI has slightly stronger termination guarantees in presence of multiple fixed points, and works in practice for max. probabilities without collapsing end components despite the lack of a guarantee. Like II, it can be combined with alternative methods for dealing with end components such as the new *deflating* technique of [23]. OVI is a *simple* algorithm that is *easy* to add to any tool that already implements value iteration, and it is *fast*, further closing the performance gap between VI and sound methods.

Acknowledgments. The authors thank Tim Quatmann (RWTH Aachen) for fruitful discussions when the idea of OVI initially came up in late 2018, and for his help in implementing and optimising the SVI implementation in mcsta.

Data Availability. A dataset to replicate our experimental evaluation is archived and available at DOI 10.4121/uuid:3df859e6-edc6-4e2d-92f3-93e478bbe8dc [19].

References

1. Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science, vol. 3, pp. 1–168. Oxford University Press (1994). http://www.cs.bham.ac.uk/~axj/pub/papers/handy1.pdf (corrected and expanded version)
2. Ashok, P., Křetínský, J., Weininger, M.: PAC statistical model checking for Markov decision processes and stochastic games. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 497–519. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_29
3. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for Markov decision processes. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 160–180. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_8
4. Balaji, N., Kiefer, S., Novotný, P., Pérez, G.A., Shirmohammadi, M.: On the complexity of value iteration. In: 46th International Colloquium on Automata, Languages, and Programming (ICALP). LIPIcs, vol. 132, pp. 102:1–102:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ICALP.2019.102
5. Bauer, M.S., Mathur, U., Chadha, R., Sistla, A.P., Viswanathan, M.: Exact quantitative probabilistic model checking through rational search. In: FMCAD, pp. 92–99. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102246
6. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60692-0_70
7. 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
8. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. TACAS. LNCS **10206**, 151–168 (2017). https://doi.org/10.1007/978-3-662-54580-5_9
9. Chatterjee, K., Henzinger, T.A.: Value iteration. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0_7
10. 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
11. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS, pp. 342–351. IEEE Computer Society (2010). https://doi.org/10.1109/LICS.2010.41
12. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_3
13. Fränzle, M., Hahn, E.M., Hermanns, H., Wolovick, N., Zhang, L.: Measurability and safety verification for stochastic hybrid systems. In: HSCC, pp. 43–52. ACM (2011). https://doi.org/10.1145/1967701.1967710

14. Haddad, S., Monmege, B.: Reachability in MDPs: refining convergence of value iteration. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) RP 2014. LNCS, vol. 8762, pp. 125–137. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11439-2_10
15. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018). https://doi.org/10.1016/j.tcs.2016.12.003
16. Hahn, E.M., et al.: The 2019 comparison of tools for the analysis of quantitative formal models. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 69–92. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_5
17. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.P.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2013). https://doi.org/10.1007/s10703-012-0167-z
18. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: a web-based probabilistic model checker. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 312–317. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_22
19. Hartmanns, A.: Optimistic value iteration (artifact). 4TU.Centre for Research Data (2019). https://doi.org/10.4121/uuid:3df859e6-edc6-4e2d-92f3-93e478bbe8dc
20. 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
21. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 344–350. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_20
22. Hensel, C.: The probabilistic model checker Storm: symbolic methods for probabilistic model checking. Ph.D. thesis, RWTH Aachen University, Germany (2018)
23. Kelmendi, E., Krämer, J., Křetínský, J., Weininger, M.: Value iteration for simple stochastic games: stopping criterion and learning algorithm. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 623–642. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_36
24. 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
25. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods Syst. Des. **29**(1), 33–78 (2006). https://doi.org/10.1007/s10703-006-0005-2
26. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. **282**(1), 101–150 (2002). https://doi.org/10.1016/S0304-3975(01)00046-9
27. Lassez, J.L., Nguyen, V.L., Sonenberg, L.: Fixed point theorems and semantics: a folk tale. Inf. Process. Lett. **14**(3), 112–116 (1982)
28. McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: ICML, ACM International Conference Proceeding Series, vol. 119, pp. 569–576. ACM (2005). https://doi.org/10.1145/1102351.1102423

29. Park, D.: Fixpoint induction and proofs of program properties. Mach. Intell. **5** (1969)
30. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Mathematical Statistics: Applied Probability and Statistics. Wiley, New York (1994)
31. Quatmann, T., Katoen, J.-P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 643–661. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_37

PrIC3: Property Directed Reachability for MDPs

Kevin Batz[1(✉)], Sebastian Junges[2], Benjamin Lucien Kaminski[3], Joost-Pieter Katoen[1], Christoph Matheja[4], and Philipp Schröer[1]

[1] RWTH Aachen University, Aachen, Germany
kevin.batz@cs.rwth-aachen.de
[2] University of California, Berkeley, USA
[3] University College London, London, UK
[4] ETH Zürich, Zürich, Switzerland

Abstract. IC3 has been a leap forward in symbolic model checking. This paper proposes PrIC3 (pronounced pricy-three), a conservative extension of IC3 to symbolic model checking of MDPs. Our main focus is to develop the theory underlying PrIC3. Alongside, we present a first implementation of PrIC3 including the key ingredients from IC3 such as generalization, repushing, and propagation.

1 Introduction

IC3. Also known as property-directed reachability (PDR) [23], IC3 [13] is a symbolic approach for verifying finite transition systems (TSs) against safety properties like "*bad states are unreachable*". It combines bounded model checking (BMC) [12] and inductive invariant generation. Put shortly, IC3 either proves that a set B of bad states is *un*reachable by finding a set of non-B states closed under reachability—called an *inductive invariant*—or refutes reachability of B by a *counterexample* path reaching B. Rather than unrolling the transition relation (as in BMC), IC3 attempts to incrementally strengthen the invariant "no state in B is reachable" into an inductive one. In addition, it applies aggressive abstraction to the explored state space, so-called generalization [36]. These aspects together with the enormous advances in modern SAT solvers have led to IC3's success. IC3 has been extended [27,38] and adapted to software verification [19,44]. This paper develops a *quantitative* IC3 framework for probabilistic models.

MDPs. Markov decision processes (MDPs) extend TSs with discrete probabilistic choices. They are central in planning, AI as well as in modeling randomized distributed algorithms. A key question in verifying MDPs is *quantitative* reachability: "*is the (maximal) probability to reach B at most λ?*". Quantitative reachability [5,6]

This work has been supported by the ERC Advanced Grant 787914 (FRAPPANT), NSF grants 1545126 (VeHICaL) and 1646208, the DARPA Assured Autonomy program, Berkeley Deep Drive, and by Toyota under the iCyPhy center.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 512–538, 2020.
https://doi.org/10.1007/978-3-030-53291-8_27

reduces to solving linear programs (LPs). Various tools support MDP model checking, e.g., Prism [43], Storm [22], modest [34], and EPMC [31]. The LPs are mostly solved using (variants of) value iteration [8,28,35,51]. Symbolic BDD-based MDP model checking originated two decades ago [4] and is rather successful.

Towards IC3 *for MDPs.* Despite the success of BDD-based symbolic methods in tools like Prism, IC3 has not penetrated probabilistic model checking yet. The success of IC3 and the importance of quantitative reachability in probabilistic model checking raises the question *whether and how* IC3 *can be adapted—not just utilized—to reason about quantitative reachability in MDPs.* This paper addresses the challenges of answering this question. It extends IC3 in several dimensions to overcome these hurdles, making PrIC3—to our knowledge—*the first* IC3 *framework for quantitative reachability in MDPs*[1]. Notably, PrIC3 is conservative: For a threshold $\lambda = 0$, PrIC3 solves the same qualitative problem *and behaves (almost) the same as standard* IC3. Our main contribution is developing the theory underlying PrIC3, which is accompanied by a proof-of-concept implementation.

Challenge 1 (Leaving the Boolean domain). IC3 iteratively computes *frames*, which are over-approximations of sets of states that can reach B in a bounded number of steps. For MDPs, Boolean reachability becomes a *quantitative reachability probability.* This requires a shift: frames become real-valued functions rather than sets of states. Thus, there are infinitely many possible frames—even for finite-state MDPs—just as for infinite-state software [19,44] and hybrid systems [54]. Additionally, whereas in TSs a state reachable within k steps remains reachable on increasing k, the reachability probability in MDPs may increase. This complicates ensuring termination of an IC3 algorithm for MDPs. △

Challenge 2 (Counterexamples ≠ single paths). For TSs, a single cycle-free path[2] to B suffices to refute that "B *is not reachable*". This is not true in the probabilistic setting [32]. Instead, proving that the probability of reaching B exceeds the threshold λ requires *a set of possibly cyclic paths*—e.g., represented as a sub-MDP [15]—whose probability mass exceeds λ. Handling sets of paths as counterexamples in the context of IC3 is new. △

Challenge 3 (Strengthening). This key IC3 technique intuitively turns a proof obligation of type (I) "state s is unreachable from the initial state s_I" into type (II) "s's *predecessors* are unreachable from s_I". A first issue is that in the quantitative setting, the standard characterization of reachability probabilities in MDPs (the Bellman equations) inherently *reverses* the direction of reasoning (cf. "reverse" IC3 [53]): Hence, strengthening turns (I) "s cannot reach B" into (II) "s's *successors* cannot reach B".

A much more challenging issue, however, is that in the quantitative setting obligations of type (I) read "s is reachable *with at most probability* δ". However,

[1] Recently, (standard) IC3 for TSs was *utilized* in model checking Markov chains [49] to on-the-fly compute the states that cannot reach B.

[2] In [38], tree-like counterexamples are used for non-linear predicate transformers in IC3.

the strengthened type (II) obligation must then read: "*the weighted sum over the reachability probabilities of the successors of* s is at most δ". In general, there are infinitely many possible choices of subobligations for the successors of s in order to satisfy the original obligation, because—grossly simplified—there are infinitely many possibilities for a and b to satisfy weighted sums such as $\frac{1}{3}a + \frac{2}{3}b \leq \delta$. While we only need one choice of subobligations, picking a *good* one is approximately as hard as solving the entire problem altogether. We hence require a heuristic, which is guided by a *user-provided oracle*. △

Challenge 4 (Generalization). "One of the key components of IC3 is [inductive] generalization" [13]. Generalization [36] abstracts single states. It makes IC3 scale, but is *not* essential for correctness. To facilitate generalization, systems should be encoded symbolically, i.e., integer-valued program variables describe states. Frames thus map variables to probabilities. A first aspect is how to effectively present them to an SMT-solver. Conceptually, we use uninterpreted functions and universal quantifiers (encoding program behavior) together with linear real arithmetic to encode the weighted sums occurring when reasoning about probabilities. A second aspect is more fundamental: Abstractly, IC3's generalization guesses an unreachable set of states. We, however, need to guess this set *and* a probability for each state. To be effective, these guesses should moreover eventually yield an inductive frame, which is often highly nonlinear. We propose three SMT-guided interpolation variants for guessing these maps. △

Structure of this Paper. We develop PrIC3 gradually: We explain the underlying rationale in Sect. 3. We also describe the core of PrIC3—called $\text{PrIC3}_{\mathcal{H}}$—which resembles closely the main loop of standard IC3, but uses adapted frames and termination criteria (Challenge 1). In line with Challenge 3, $\text{PrIC3}_{\mathcal{H}}$ is parameterized by a heuristic $\mathcal{H}$ which is applied whenever we need to select one out of infinitely many probabilities. No requirements on the quality of $\mathcal{H}$ are imposed. $\text{PrIC3}_{\mathcal{H}}$ is *sound* and always terminates: If it returns `true`, then the maximal reachability probability is bounded by λ. Without additional assumptions about $\mathcal{H}$, $\text{PrIC3}_{\mathcal{H}}$ is *incomplete*: on returning `false`, it is unknown whether the returned sub-MDP is indeed a counterexample (Challenge 2). Section 4 details strengthening (Challenge 3). Section 5 presents a sound *and* complete algorithm PrIC3 on top of $\text{PrIC3}_{\mathcal{H}}$. Section 6 presents a prototype, discusses our chosen heuristics, and addresses Challenge 4. Section 7 shows some encouraging experiments, but also illustrates need for further progress.

Related Work. Just like IC3 has been a symbiosis of different approaches, PrIC3 has been inspired by several existing techniques from the verification of probabilistic systems.

BMC. Adaptions of BMC to Markov chains (MCs) with a dedicated treatment of cycles have been pursued in [57]. The encoding in [24] annotates sub-formulae with probabilities. The integrated SAT solving process implicitly unrolls all paths leading to an exponential blow-up. In [52], this is circumvented by grouping paths, discretizing them, and using an encoding with quantifiers and bit-vectors,

but without numerical values. Recently, [56] extends this idea to a PAC algorithm by purely propositional encodings and (approximate) model counting [17]. These approaches focus on MCs and are not mature yet.

Invariant Synthesis. Quantitative loop invariants are key in analyzing *probabilistic programs* whose operational semantics are (possibly infinite) MDPs [26]. A quantitative invariant I maps states to probabilities. I is shown to be an invariant by comparing I to the result of applying the MDP's Bellman operator to I. Existing approaches for invariant synthesis are, e.g., based on weakest pre-expectations [33,39,40,42,46], template-based constraint solving [25], notions of martingales [3,9,16,55], and solving recurrence relations [10]. All but the last technique require user guidance.

Abstraction. To combat state-space explosion, abstraction is often employed. CEGAR for MDPs [37] deals with explicit sets of paths as counterexamples. Game-based abstraction [30,41] and partial exploration [14] exploit that not all paths have to be explored to prove bounds on reachability probabilities.

Statistical Methods and (deep) Reinforcement Learning. Finally, an avenue that avoids storing a (complete) model are simulation-based approaches (statistical model checking [2]) and variants of reinforcement learning, possibly with neural networks. For MDPs, these approaches yield weak statistical guarantees [20], but may provide good oracles.

2 Problem Statement

Our aim is to prove that the *maximal probability* of *reaching* a *set B of bad states* from the initial state s_I of a *Markov decision process* $\mathfrak{M}$ is at most some *threshold* λ. Below, we give a formal description of our problem. We refer to [7,50] for a thorough introduction.

Definition 1 (MDPs). *A* Markov decision process *(*MDP*) is a tuple* $\mathfrak{M} = (S, s_I, \mathrm{Act}, P)$*, where S is a finite set of* states*, $s_I \in S$ is the* initial state*,* Act *is a finite set of* actions*, and $P\colon S \times \mathrm{Act} \times S \to [0,1]$ is a* transition probability function*. For state s, let $\mathrm{Act}(s) = \{a \in \mathrm{Act} \mid \exists s' \in S\colon P(s,a,s') > 0\}$ be the* enabled actions *at s. For all states $s \in S$, we require $|\mathrm{Act}(s)| \geq 1$ and $\sum_{s' \in S} P(s,a,s') = 1$.* △

For this paper, we fix an MDP $\mathfrak{M} = (S, s_I, \mathrm{Act}, P)$, a set of *bad states* $B \subseteq S$, and a threshold $\lambda \in [0,1]$. The *maximal*[3] *(unbounded) reachability probability* to eventually reach a state in B from a state s is denoted by $\Pr^{\max}(s \models \Diamond B)$. We characterize $\Pr^{\max}(s \models \Diamond B)$ using the so-called *Bellman operator*. Let M^N denote the set of functions from N to M. Anticipating IC3 terminology, we call a function $F \in [0,1]^S$ a *frame*. We denote by $F[s]$ the evaluation of frame F for state s.

[3] Maximal with respect to all possible resolutions of nondeterminism in the MDP.

Definition 2 (Bellman Operator). *For a set of actions $A \subseteq \mathrm{Act}$, we define the* Bellman operator *for A as a frame transformer $\Phi_A \colon [0,1]^S \to [0,1]^S$ with*

$$\Phi_A(F)[s] \;=\; \begin{cases} 1, & \text{if } s \in B \\ \max\limits_{a \in A} \sum\limits_{s' \in S} P(s,a,s') \cdot F[s'] \;, & \text{if } s \notin B \;. \end{cases}$$

We write Φ_a for $\Phi_{\{a\}}$, Φ for Φ_{Act}, and call Φ simply the Bellman operator. △

For every state s, the maximal reachability probability $\Pr^{\max}(s \models \Diamond B)$ is then given by the least fixed point of the Bellman operator Φ. That is,

$$\forall\, s\colon \quad \Pr^{\max}(s \models \Diamond B) \;=\; \big(\mathrm{lfp}\ \Phi\big)[s] \;,$$

where the underlying partial order on frames is a complete lattice with ordering

$$F_1 \;\leq\; F_2 \qquad \text{iff} \qquad \forall\, s \in S\colon \quad F_1[s] \;\leq\; F_2[s] \;.$$

In terms of the Bellman operator, our formal problem statement reads as follows:

Given an MDP $\mathfrak{M}$ with initial state s_I, a set B of bad states, and a *threshold* $\lambda \in [0,1]$,
prove or refute that $\quad \Pr^{\max}(s_I \models \Diamond B) \;=\; \big(\mathrm{lfp}\ \Phi\big)[s_I] \quad \leq \quad \lambda \;.$

Whenever $\Pr^{\max}(s_I \models \Diamond B) \leq \lambda$ indeed holds, we say that the MDP $\mathfrak{M}$ is *safe* (with respect to the set of bad states B and threshold λ); otherwise, we call it *unsafe*.

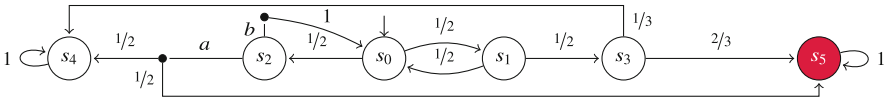


Fig. 1. The MDP $\mathfrak{M}$ serving as a running example.

Recovery Statement 1. *For $\lambda = 0$, our problem statement is equivalent to the* qualitative reachability *problem solved by (reverse) standard* IC3*, i.e, prove or refute that all bad states in B are* unreachable *from the initial state s_I.*

Example 1. The MDP $\mathfrak{M}$ in Fig. 1 consists of 6 states with initial state s_0 and bad states $B = \{s_5\}$. In s_2, actions a and b are enabled; in all other states, one unlabeled action is enabled. We have $\Pr^{\max}(s_0 \models \Diamond B) = 2/3$. Hence, $\mathfrak{M}$ is safe for all thresholds $\lambda \geq 2/3$ and unsafe for $\lambda < 2/3$. In particular, $\mathfrak{M}$ is unsafe for $\lambda = 0$ as s_5 is *reachable* from s_0. △

3 The Core PrIC3 Algorithm

The purpose of PrIC3 is to prove or refute that the maximal probability to reach a bad state in B from the initial state s_I of the MDP $\mathfrak{M}$ is at most λ. In this section, we explain the rationale underlying PrIC3. Moreover, we describe the core of PrIC3—called PrIC3$_{\mathcal{H}}$—which bears close resemblance to the main loop of standard IC3 for TSs.

Because of the inherent direction of the Bellman operator, we build PrIC3 on *reverse* IC3 [53], cf. Challenge 3. Reversing constitutes a shift from reasoning along the direction *initial-to-bad* to *bad-to-initial*. While this shift is mostly *inessential* to the fundamentals underlying IC3, the reverse direction is unswayable in the probabilistic setting. Whenever we draw a connection to standard IC3, we thus generally mean *reverse* IC3.

3.1 Inductive Frames

IC3 for TSs operates on (*quali*tative) frames representing sets of states of the TS at hand. A frame F can hence be thought of as a mapping[4] from states to $\{0,1\}$. In PrIC3 for MDPs, we need to move from a Boolean to a quantitative regime. Hence, a (*quanti*tative) frame is a mapping from states to probabilities in $[0,1]$.

For a given TS, consider the frame transformer T that adds to a given input frame F' all bad states in B and all predecessors of the states contained in F'. The rationale of standard (reverse) IC3 is to find a frame $F \in \{0,1\}^S$ such that (I) the initial state s_I does not belong to F and (II) applying T takes us down in the partial order on frames, i.e.,

$$\text{(I)} \quad F[s_I] \;=\; 0 \qquad \text{and} \qquad \text{(II)} \quad T(F) \;\leq\; F \;.$$

Intuitively, (I) postulates the *hypothesis* that s_I cannot reach B and (II) expresses that F is closed under adding bad states and taking predecessors, thus affirming the hypothesis.

Analogously, the rationale of PrIC3 is to find a frame $F \in [0,1]^S$ such that (I) F postulates that the probability of s_I to reach B is at most the threshold λ and (II) applying the Bellman operator Φ to F takes us down in the partial order on frames, i.e.,

$$\text{(I)} \quad F[s_I] \;\leq\; \lambda \qquad \text{and} \qquad \text{(II)} \quad \Phi(F) \;\leq\; F \;.$$

Frames satisfying the above conditions are called *inductive invariants* in IC3. We adopt this terminology. By *Park's Lemma* [48], which in our setting reads

$$\Phi(F) \;\leq\; F \quad \text{implies} \quad \text{lfp}\ \Phi \;\leq\; F \;,$$

[4] In IC3, frames are typically characterized by logical formulae. To understand IC3's fundamental principle, however, we prefer to think of frames as functions in $\{0,1\}^S$ partially ordered by $\leq$.

an inductive invariant F would indeed *witness* that $\Pr^{\max}(s_I \models \Diamond B) \leq \lambda$, because

$$\Pr^{\max}(s_I \models \Diamond B) \;=\; \big(\mathrm{lfp}\ \Phi\big)[s_I] \;\leq\; F[s_I] \;\leq\; \lambda\,.$$

If no inductive invariant exists, then standard IC3 will find a counterexample: a *path* from the initial state s_I to a bad state in B, which serves as a witness to refute. Analogously, PrIC3 will find a counterexample, but of a different kind: Since single paths are insufficient as counterexamples in the probabilistic realm (Challenge 2), PrIC3 will instead find a *subsystem* of states of the MDP witnessing $\Pr^{\max}(s_I \models \Diamond B) > \lambda$.

3.2 The PrIC3 Invariants

Analogously to standard IC3, PrIC3 aims to find the inductive invariant by maintaining a *sequence of frames* $F_0 \leq F_1 \leq F_2 \leq \ldots$ such that $F_i[s]$ overapproximates the maximal probability of reaching B from s within *at most* i *steps*. This *i-step-bounded reachability probability* $\Pr^{\max}\left(s \models \Diamond^{\leq i} B\right)$ can be characterized using the Bellman operator: $\Phi(\mathbf{0})$ is the 0-step probability; it is 1 for every $s \in B$ and 0 otherwise. For any $i \geq 0$, we have

$$\Pr^{\max}\left(s \models \Diamond^{\leq i} B\right) \;=\; \Big(\Phi^i\left(\Phi\left(\mathbf{0}\right)\right)\Big)[s] \;=\; \Big(\Phi^{i+1}\left(\mathbf{0}\right)\Big)[s]\,,$$

where $\mathbf{0}$, the frame that maps every state to 0, is the least frame of the underlying complete lattice. For a finite MDP, the *unbounded* reachability probability is then given by the limit

$$\Pr^{\max}(s \models \Diamond B) \;=\; \big(\mathrm{lfp}\ \Phi\big)[s] \;\overset{(*)}{=}\; \Big(\lim_{n\to\infty} \Phi^n\left(\mathbf{0}\right)\Big)[s] \;=\; \lim_{n\to\infty} \Pr^{\max}\Big(s \models \Diamond^{\leq n} B\Big)\,,$$

where $(*)$ is a consequence of the well-known Kleene fixed point theorem [45].

The sequence $F_0 \leq F_1 \leq F_2 \leq \ldots$ maintained by PrIC3 should frame-wise overapproximate the increasing sequence $\Phi(\mathbf{0}) \leq \Phi^2(\mathbf{0}) \leq \Phi^3(\mathbf{0}) \ldots$. Pictorially:

$$\begin{array}{ccccccccccc}
 & & F_0 & \leq & F_1 & \leq & F_2 & \leq & \ldots & \leq & F_k \\
 & & \vee| & & \vee| & & \vee| & & & & \vee| \\
\mathbf{0} & \leq & \Phi(\mathbf{0}) & \leq & \Phi^2(\mathbf{0}) & \leq & \Phi^3(\mathbf{0}) & \leq & \ldots & \leq & \Phi^{k+1}(\mathbf{0})
\end{array}$$

However, the sequence $\Phi(\mathbf{0}), \Phi^2(\mathbf{0}), \Phi^3(\mathbf{0}), \ldots$ will never explicitly be known to PrIC3. Instead, PrIC3 will ensure the above frame-wise overapproximation property implicitly by enforcing the so-called PrIC3 *invariants* on the frame sequence $F_0, F_1, F_2, \ldots$. Apart from allowing for a threshold $0 \leq \lambda \leq 1$ on the maximal reachability probability, these invariants coincide with the standard IC3 invariants (where $\lambda = 0$ is fixed). Formally:

Definition 3 (PrIC3 Invariants). *Frames $F_0, \ldots, F_k$, for $k \geq 0$, satisfy the* PrIC3 *invariants, a fact we will denote by* $\mathsf{PrIC3Inv}(F_0, \ldots, F_k)$*, if all of the following hold:*

1.	**Initiality** :	$F_0 = \Phi(\mathbf{0})$	
2.	**Chain Property** :	$\forall\, 0 \leq i < k\colon$	$F_i \leq F_{i+1}$
3.	**Frame-safety** :	$\forall\, 0 \leq i \leq k\colon$	$F_i[s_I] \leq \lambda$
4.	**Relative Inductivity** :	$\forall\, 0 \leq i < k\colon$	$\Phi(F_i) \leq F_{i+1}$

△

The PrIC3 invariants enforce the above picture: The *chain property* ensures $F_0 \leq F_1 \leq \ldots \leq F_k$. We have $\Phi(\mathbf{0}) = F_0 \leq F_0$ by *initiality*. Assuming $\Phi^{i+1}(\mathbf{0}) \leq F_i$ as induction hypothesis, monotonicity of Φ and *relative inductivity* imply $\Phi^{i+2}(\mathbf{0}) \leq \Phi(F_i) \leq F_{i+1}$.

By overapproximating $\Phi(\mathbf{0}), \Phi^2(\mathbf{0}), \ldots, \Phi^{k+1}(\mathbf{0})$, the frames $F_0, \ldots, F_k$ in effect bound the maximal step-bounded reachability probability of every state:

Lemma 1. *Let frames $F_0, \ldots, F_k$ satisfy the* PrIC3 *invariants. Then*

$$\forall\, s \;\; \forall\, i \leq k\colon \quad Pr^{max}\left(s \models \Diamond^{\leq i} B\right) \;\leq\; F_i[s].$$

In particular, Lemma 1 together with *frame-safety* ensures that the maximal step-bounded reachability probability of the *initial state* s_I to reach B is at most the threshold λ.

As for proving that the *unbounded* reachability probability is also at most λ, it suffices to find two consecutive frames, say F_i and F_{i+1}, that coincide:

Lemma 2. *Let frames $F_0, \ldots, F_k$ satisfy the* PrIC3 *invariants. Then*

$$\exists\, i < k\colon \quad F_i = F_{i+1} \qquad \textit{implies} \qquad Pr^{max}(s_I \models \Diamond B) \;\leq\; \lambda\,.$$

Proof. $F_i = F_{i+1}$ and *relative inductivity* yield $\Phi(F_i) \leq F_{i+1} = F_i$, rendering F_i *inductive*. By Park's lemma (cf. Sect. 3.1), we obtain $\mathrm{lfp}\ \Phi \leq F_i$ and—by *frame-safety*—conclude

$$\Pr^{\max}(s_I \models \Diamond B) \;=\; \big(\mathrm{lfp}\ \Phi\big)[s_I] \;\leq\; F_i[s_I] \;\leq\; \lambda\,. \qquad \square$$

3.3 Operationalizing the PrIC3 Invariants for Proving Safety

Lemma 2 gives us a clear angle of attack for *proving* an MDP safe: Repeatedly add and refine frames approximating step-bounded reachability probabilities for more and more steps while enforcing the PrIC3 invariants (cf. Definition 3.2) until two consecutive frames coincide.

Analogously to standard IC3, this approach is taken by the core loop $\mathsf{PrIC3}_{\mathcal{H}}$ depicted in Algorithm 1; differences to the main loop of IC3 (cf. [23, Fig. 5]) are highlighted in red. A particular difference is that $\mathsf{PrIC3}_{\mathcal{H}}$ is parameterized by a heuristic $\mathcal{H}$ for finding suitable probabilities (see Challenge 3). Since the precise choice of $\mathcal{H}$ is irrelevant for the soundness of $\mathsf{PrIC3}_{\mathcal{H}}$, we defer a detailed discussion of suitable heuristics to Sect. 4.

Data: MDP $\mathfrak{M}$, set of bad states B, threshold λ
Result: `true` or `false` and a subset of the states of $\mathfrak{M}$
1 $F_0 \leftarrow \Phi(\mathbf{0})$; $F_1 \leftarrow \mathbf{1}$; $k \leftarrow 1$; $\mathit{oldSubsystem} \leftarrow \emptyset$;
2 **while** `true` **do**
3 $\quad \mathit{success}, F_0, \dots, F_k, \mathit{subsystem} \leftarrow \mathsf{Strengthen}_{\mathcal{H}}(F_0, \dots, F_k)$;
4 $\quad$ **if** $\neg \mathit{success}$ **then return** `false`, $\mathit{subsystem}$;
5 $\quad F_{k+1} \leftarrow \mathbf{1}$;
6 $\quad F_0, \dots, F_{k+1} \leftarrow \mathsf{Propagate}(F_0, \dots, F_{k+1})$;
7 $\quad$ **if** $\exists\, 1 \leq i \leq k \colon F_i = F_{i+1}$ **then return** `true`, _;
8 $\quad$ **if** $\mathit{oldSubsystem} = \mathit{subsystem}$ **then return** `false`, $\mathit{subsystem}$;
9 $\quad k \leftarrow k+1$; $\mathit{oldSubsystem} \leftarrow \mathit{subsystem}$;
10 **end**

Algorithm 1: $\mathsf{PrIC3}_{\mathcal{H}}(\mathfrak{M}, B, \lambda)$

As input, $\mathsf{PrIC3}_{\mathcal{H}}$ takes an MDP $\mathfrak{M} = (S, s_I, \mathrm{Act}, P)$, a set $B \subseteq S$ of bad states, and a threshold $\lambda \in [0, 1]$. Since the input is never changed, we assume it to be *globally available*, also to subroutines. As output, $\mathsf{PrIC3}_{\mathcal{H}}$ returns `true` if two consecutive frames become equal. We hence say that $\mathsf{PrIC3}_{\mathcal{H}}$ is *sound* if it only returns `true` if $\mathfrak{M}$ is safe.

We will formalize soundness using Hoare triples. For precondition ϕ, postcondition ψ, and program P, the triple $\{\, \phi \,\}\, P \,\{\, \psi \,\}$ is *valid* (for partial correctness) if, whenever program P starts in a state satisfying precondition ϕ and terminates in some state s', then s' satisfies postcondition ψ. Soundness of $\mathsf{PrIC3}_{\mathcal{H}}$ then means validity of the triple

$$\{\, \texttt{true} \,\}\ \mathit{safe}, \_ \leftarrow \mathsf{PrIC3}_{\mathcal{H}}(\mathfrak{M}, B, \lambda)\ \{\, \mathit{safe} \Rightarrow \Pr^{\max}(s_I \models \Diamond B) \leq \lambda \,\}\,.$$

Let us briefly go through the individual steps of $\mathsf{PrIC3}_{\mathcal{H}}$ in Algorithm 1 and convince ourselves that it is indeed sound. After that, we discuss why $\mathsf{PrIC3}_{\mathcal{H}}$ terminates and what happens if it is unable to prove safety by finding two equal consecutive frames.

How $\mathsf{PrIC3}_{\mathcal{H}}$ works. Recall that $\mathsf{PrIC3}_{\mathcal{H}}$ maintains a sequence of frames $F_0, \dots, F_k$ which is initialized in l. 1 with $k = 1$, $F_0 = \Phi(\mathbf{0})$, and $F_1 = \mathbf{1}$, where the frame $\mathbf{1}$ maps every state to 1. Every time upon entering the **while**-loop in terms l. 2, the initial segment $F_0, \dots, F_{k-1}$ satisfies all $\mathsf{PrIC3}$ invariants (cf. Definition 3), whereas the full sequence $F_0, \dots, F_k$ potentially violates frame-safety as it is possible that $F_k[s_I] > \lambda$.

In l. 3, procedure $\mathsf{Strengthen}_{\mathcal{H}}$—detailed in Sect. 4—is called to restore *all* $\mathsf{PrIC3}$ invariants on the *entire* frame sequence: It either returns `true` if successful or returns `false` and a counterexample (in our case a subsystem of the MDP) if it was unable to do so. To ensure soundness of $\mathsf{PrIC3}_{\mathcal{H}}$, it suffices that $\mathsf{Strengthen}_{\mathcal{H}}$ restores the $\mathsf{PrIC3}$ invariants whenever it returns `true`. Formally, $\mathsf{Strengthen}_{\mathcal{H}}$ must meet the following specification:

Definition 4. *Procedure* $\mathsf{Strengthen}_{\mathcal{H}}$ *is* sound *if the following Hoare triple is valid:*

$$\begin{aligned}&\left\{\, \mathsf{PrIC3Inv}\,(F_0,\ldots,F_{k-1}) \;\wedge\; F_{k-1} \leq F_k \;\wedge\; \Phi\,(F_{k-1}) \leq F_k \,\right\}\\ &\qquad \mathit{success},\, F_0,\ldots,F_k,\, \_\_ \;\leftarrow\; \mathsf{Strengthen}_{\mathcal{H}}\,(F_0,\,\ldots,\,F_k)\\ &\left\{\, \mathit{success} \;\Rightarrow\; \mathsf{PrIC3Inv}\,(F_0,\ldots,F_k) \,\right\}.\end{aligned}$$

If $\mathsf{Strengthen}_{\mathcal{H}}$ returns `true`, then a new frame $F_{k+1} = \mathbf{1}$ is created in l. 5. After that, the (now initial) segment $F_0, \ldots, F_k$ again satisfies all PrIC3 invariants, whereas the full sequence $F_0, \ldots, F_{k+1}$ potentially violates frame-safety at F_{k+1}. *Propagation* (l. 6) aims to speed up termination by updating $F_{i+1}[s]$ by $F_i[s]$ iff this does not violate relative inductivity. Consequently, the previously mentioned properties remain unchanged.

If $\mathsf{Strengthen}_{\mathcal{H}}$ returns `false`, the PrIC3 invariants—premises to Lemma 2 for witnessing safety—cannot be restored and $\mathsf{PrIC3}_{\mathcal{H}}$ terminates returning `false` (l. 4). Returning `false` (also possible in l. 8) has by specification no affect on soundness of $\mathsf{PrIC3}_{\mathcal{H}}$.

In l. 7, we check whether there exist two identical consecutive frames. If so, Lemma 2 yields that the MDP is safe; consequently, $\mathsf{PrIC3}_{\mathcal{H}}$ returns `true`. Otherwise, we increment k and are in the same setting as upon entering the loop, now with an increased frame sequence; $\mathsf{PrIC3}_{\mathcal{H}}$ then performs another iteration. In summary, we obtain:

Theorem 1 (Soundness of $\mathsf{PrIC3}_{\mathcal{H}}$). *If* $\mathsf{Strengthen}_{\mathcal{H}}$ *is sound and* Propagate *does not affect the* PrIC3 *invariants, then* $\mathsf{PrIC3}_{\mathcal{H}}$ *is sound, i.e., the following triple is valid:*

$$\left\{\, \mathtt{true} \,\right\} \mathit{safe}, \_\_ \;\leftarrow\; \mathsf{PrIC3}_{\mathcal{H}}\,(\mathfrak{M},\, B,\, \lambda)\; \left\{\, \mathit{safe} \;\Longrightarrow\; Pr^{max}\,(s_I \models \Diamond B) \;\leq\; \lambda \,\right\}$$

$\mathsf{PrIC3}_{\mathcal{H}}$ Terminates for Unsafe MDPs. If the MDP is unsafe, then there exists a step-bound n, such that $\Pr^{\max}\left(s_I \models \Diamond^{\leq n} B\right) > \lambda$. Furthermore, any sound implementation of $\mathsf{Strengthen}_{\mathcal{H}}$ (cf. Definition 4) either immediately terminates $\mathsf{PrIC3}_{\mathcal{H}}$ by returning `false` or restores the PrIC3 invariants for $F_0, \ldots, F_k$. If the former case never arises, then $\mathsf{Strengthen}_{\mathcal{H}}$ will eventually restore the PrIC3 invariants for a frame sequence of length $k = n$. By Lemma 1, we have $F_n[s_I] \geq \Pr^{\max}\left(s_I \models \Diamond^{\leq n} B\right) > \lambda$ contradicting frame-safety.

$\mathsf{PrIC3}_{\mathcal{H}}$ Terminates for Safe MDPs. Standard IC3 terminates on safe finite TSs as there are only finitely many different frames, making every ascending chain of frames eventually stabilize. For us, frames map states to probabilities (Challenge 1), yielding *infinitely many possible frames* even for finite MDPs. Hence, $\mathsf{Strengthen}_{\mathcal{H}}$ need not ever yield a stabilizing chain of frames. If it continuously fails to stabilize while repeatedly reasoning about the same set of states, we give up. $\mathsf{PrIC3}_{\mathcal{H}}$ checks this by comparing the subsystem $\mathsf{Strengthen}_{\mathcal{H}}$ operates on with the one it operated on in the previous loop iteration (l. 8).

Theorem 2. *If* $\mathsf{Strengthen}_{\mathcal{H}}$ *and* Propagate *terminate, then* $\mathsf{PrIC3}_{\mathcal{H}}$ *terminates.*

Recovery Statement 2. *For qual. reachability ($\lambda = 0$),* $\mathsf{PrIC3}_{\mathcal{H}}$ *never terminates in l. 8.*

$\mathsf{PrIC3}_{\mathcal{H}}$ is Incomplete. Standard IC3 either proves safety or returns `false` and a counterexample—a single path from the initial to a bad state. As single paths are insufficient as counterexamples in MDPs (Challenge 2), $\mathsf{PrIC3}_{\mathcal{H}}$ instead returns a *subsystem* of the MDP $\mathfrak{M}$ provided by $\mathsf{Strengthen}_{\mathcal{H}}$. However, as argued above, we cannot trust $\mathsf{Strengthen}_{\mathcal{H}}$ to provide a stabilizing chain of frames. Reporting `false` thus only means that the given MDP *may* be unsafe; the returned subsystem has to be analyzed further.

The full PrIC3 algorithm presented in Sect. 5 addresses this issue. Exploiting the subsystem returned by $\mathsf{PrIC3}_{\mathcal{H}}$, PrIC3 returns `true` if the MDP is safe; otherwise, it returns `false` and provides a true counterexample witnessing that the MDP is unsafe.

Example 2. We conclude this section with two example executions of $\mathsf{PrIC3}_{\mathcal{H}}$ on a simplified version of the MDP in Fig. 1. Assume that action b has been removed. Then, for every state, exactly one action is enabled, i.e., we consider a Markov chain. Figure 2 depicts the frame sequences computed by $\mathsf{PrIC3}_{\mathcal{H}}$ (for a reasonable $\mathcal{H}$) on that Markov chain for two thresholds: $5/9 = \Pr^{\max}(s_0 \models \Diamond B)$ and $9/10$. In particular, notice that *proving the coarser bound of* $9/10$ *requires fewer frames than proving the exact bound of* $5/9$. △

It.	1	2		3			4				5				
F_i	F_1	F_1	F_2	F_1	F_2	F_3	F_1	F_2	F_3	F_4	F_1	F_2	F_3	F_4	F_5
s_0	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$	$5/9$
s_1	1	$11/18$	1	$11/18$	$11/18$	1	$11/18$	$11/18$	$11/18$	1	$11/18$	$11/18$	$11/18$	$11/18$	1
s_2	1	$1/2$	1	$1/2$	$1/2$	1	$1/2$	$1/2$	$1/2$	1	$1/2$	$1/2$	$1/2$	$1/2$	1
s_3	1	1	1	$2/3$	1	1	$2/3$	$2/3$	1	1	$2/3$	$2/3$	$2/3$	1	1
s_4	1	1	1	1	1	1	0	1	1	1	0	0	1	1	1
s_5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

(a) Threshold $\lambda = 5/9$

It.	1	2		3			4			
F_i	F_1	F_1	F_2	F_1	F_2	F_3	F_1	F_2	F_3	F_4
s_0	$9/10$	$9/10$	$9/10$	$9/10$	$9/10$	$9/10$	$9/10$	$9/10$	$9/10$	$9/10$
s_1	1	$99/100$	1	$99/100$	$99/100$	1	$99/100$	$99/100$	$99/100$	1
s_2	1	$81/100$	1	$81/100$	$81/100$	1	$81/100$	$81/100$	$81/100$	1
s_3	1	1	1	1	1	1	1	1	1	1
s_4	1	1	1	0	1	1	0	0	1	1
s_5	1	1	1	1	1	1	1	1	1	1

(b) Threshold $\lambda = 9/10$

Fig. 2. Two runs of $\mathsf{PrIC3}_{\mathcal{H}}$ on the Markov chain induced by selecting action a in Fig. 1. For every iteration, frames are recorded after invocation of $\mathsf{Strengthen}_{\mathcal{H}}$.

4 Strengthening in $\mathsf{PrIC3}_{\mathcal{H}}$

When the main loop of $\mathsf{PrIC3}_{\mathcal{H}}$ has created a new frame $F_k = \mathbf{1}$ in its previous iteration, this frame may violate frame-safety (Definition 3.3) because of $F_k[s_I] = 1 \not\leq \lambda$. The task of $\mathsf{Strengthen}_{\mathcal{H}}$ is to restore the PrIC3 invariants on *all* frames $F_0, \ldots, F_k$. To this end, our first *obligation* is to lower the value in frame $i = k$ for state $s = s_I$ to $\delta = \lambda \in [0, 1]$. We denote such an obligation by (i, s, δ). Observe that implicitly $\delta = 0$ in the qualitative case, i.e., when proving unreachability. An obligation (i, s, δ) is *resolved* by updating the values assigned to state s in *all frames* $F_1, \ldots, F_i$ to at most δ. That is, for all $j \leq i$, we set $F_j[s]$ to the minimum

1 $Q \leftarrow \{(k, s_I, \lambda)\}$;
2 **while** Q *not empty* **do**
3 $(i, s, \delta) \leftarrow Q$.popMin(); /* pop obligation with minimal frame index */
4 **if** $i = 0 \quad \vee \quad (s \in B \wedge \delta < 1)$ **then**
/* possible counterexample given by subsystem consisting of states popped from Q at some point */
5 **return** false, __, Q.touched();
/* check whether $F_i[s] \leftarrow \delta$ violates relative inductivity */
6 **if** $\exists a \in \mathrm{Act}(s) : \Phi_a(F_{i-1})[s] > \delta$ **then** for such an a
7 $\delta_1, \ldots, \delta_n \leftarrow \mathcal{H}(s, a, \delta)$;
8 $\{s_1, \ldots, s_n\} \leftarrow \mathrm{Succs}(s, a)$;
9 Q.push $((i-1, s_1, \delta_1), \ldots, (i-1, s_n, \delta_n), (i, s, \delta))$;
10 **else** /* resolve (i, s, δ) without violating relative inductivity */
11 $F_1[s] \leftarrow \min(F_1[s], \delta); \ldots; F_i[s] \leftarrow \min(F_i[s], \delta)$;
12 **end**
13 (/* Q empty; all obligations have been resolved */) **return** true, $F_0, \ldots, F_k$, Q.touched();

Algorithm 2: Strengthen$_{\mathcal{H}}(F_0, \ldots, F_k)$

of δ and the original value $F_j[s]$. Such an update affects neither initiality nor the chain property (Definitions 3.1, 3.2). It may, however, violate relative inductivity (Definition 3.4), i.e., $\Phi(F_{i-1}) \leq F_i$. Before resolving obligation (i, s, δ), we may thus have to further decrease some entries in F_{i-1} as well. Hence, *resolving obligations may spawn additional obligations* which have to be resolved first to maintain relative inductivity. In this section, we present a generic instance of Strengthen$_{\mathcal{H}}$ meeting its specification (Definition 4) and discuss its correctness.

Strengthen$_{\mathcal{H}}$ by Example. Strengthen$_{\mathcal{H}}$ is given by the pseudo code in Algorithm 2; differences to standard IC3 (cf. [23, Fig. 6]) are highlighted in red. Intuitively, Strengthen$_{\mathcal{H}}$ attempts to recursively resolve all obligations until either both frame-safety and relative inductivity are restored for *all* frames or it detects a *potential counterexample* justifying why it is unable to do so. We first consider an execution where the latter does not arise:

Example 3. We zoom in on Example 2: Prior to the second iteration, we have created the following three frames assigning values to the states s_0, s_5:

$$F_0 = (0, 0, 0, 0, 1), \qquad F_1 = (5/9, 1, 1, 1, 1, 1), \qquad \text{and} \qquad F_2 = \mathbf{1}.$$

To keep track of unresolved obligations (i, s, δ), Strengthen$_{\mathcal{H}}$ employs a priority queue Q which pops obligations with minimal frame index i first. Our first step is to ensure frame-safety of F_2, i.e., alter F_2 so that $F_2[s_0] \leq 5/9$; we thus initialize the queue Q with the initial obligation $(2, s_0, 5/9)$ (l. 1). To do so, we check whether updating $F_2[s_0]$ to $5/9$ would invalidate relative inductivity (l. 6). This is indeed the case:

$$\Phi\left(F_1\right)[s_0] \;=\; 1/2 \cdot F_1[s_1] + 1/2 \cdot F_1[s_2] \;=\; 1 \not\leq 5/9.$$

To restore relative inductivity, $\mathsf{Strengthen}_{\mathcal{H}}$ spawns one new obligation for each relevant successor of s_0. These have to be resolved before retrying to resolve the old obligation.[5]

In contrast to standard IC3 *, spawning obligations involves finding suitable probabilities* δ (l. 7). In our example this means we have to spawn two obligations $(1, s_1, \delta_1)$ and $(1, s_2, \delta_2)$ such that $1/2 \cdot \delta_1 + 1/2 \cdot \delta_2 \leq 5/9$. There are *infinitely many choices* for δ_1 and δ_2 satisfying this inequality. Assume some heuristic $\mathcal{H}$ chooses $\delta_1 = 11/18$ and $\delta_2 = 1/2$; we push obligations $(1, s_1, 11/18)$, $(1, s_2, 1/2)$, and $(2, s_0, 5/9)$ (ll. 8, 9). In the next iteration, we first pop obligation $(1, s_1, 11/18)$ (l. 3) and find that it can be resolved without violating relative inductivity (l. 6). Hence, we set $F_1[s_1]$ to $11/18$ (l. 11); no new obligation is spawned. Obligation $(1, s_2, 1/2)$ is resolved analogously; the updated frame is $F_1 = (5/9, 11/18, 1/2, 1)$. Thereafter, our initial obligation $(2, s_0, 5/9)$ can be resolved; relative inductivity is restored for F_0, F_1, F_2. Hence, $\mathsf{Strengthen}_{\mathcal{H}}$ returns `true` together with the updated frames. △

$\mathsf{Strengthen}_{\mathcal{H}}$ is Sound. Let us briefly discuss why Algorithm 2 meets the specification of a sound implemenation of $\mathsf{Strengthen}_{\mathcal{H}}$ (Definition 4): First, we observe that Algorithm 2 alters the frames—and thus potentially invalidates the PrIC3 invariants—only in l. 11 by resolving an obligation (i, s, δ) with $\Phi\left(F_{i-1}\right)[s] \leq \delta$ (due to the check in l. 6).
Let $F\left\langle s \mapsto \delta \right\rangle$ denote the frame F in which $F[s]$ is set to δ, i.e.,

$$F\left\langle s \mapsto \delta \right\rangle[s'] \;=\; \begin{cases} \delta, & \text{if } s' = s, \\ F[s'], & \text{otherwise.} \end{cases}$$

Indeed, resolving obligation (i, s, δ) in l. 11 lowers the values assigned to state s to at most δ *without* invalidating the PrIC3 invariants:

Lemma 3. *Let* (i, s, δ) *be an obligation and* $F_0, \ldots, F_i$*, for* $i > 0$*, be frames with* $\Phi\left(F_{i-1}\right)[s] \leq \delta$*. Then* $\mathsf{PrIC3Inv}\left(F_0, \ldots, F_i\right)$ *implies*

$$\mathsf{PrIC3Inv}\left(F_0\left\langle s \mapsto \min\left(F_0[s],\, \delta\right)\right\rangle, \ldots, F_i\left\langle s \mapsto \min\left(F_i[s],\, \delta\right)\right\rangle \right).$$

Crucially, the precondition of Definition 4 guarantees that all PrIC3 invariants except frame safety hold initially. Since these invariants are never invalidated due to Lemma 3, Algorithm 2 is a sound implementation of $\mathsf{Strengthen}_{\mathcal{H}}$ if it restores frame safety whenever it returns `true`, i.e., once it leaves the loop with an empty obligation queue Q (ll. 12–13). Now, an obligation (i, s, δ) is only popped from Q in l. 3. As (i, s, δ) is added to Q upon reaching l. 9, the size of Q can only ever be reduced (without returning `false`) by resolving (i, s, δ) in l. 11. Hence, Algorithm 2 does not return `true` unless it restored frame safety by resolving, amongst all other obligations, the initial obligation (k, s_I, λ). Consequently:

[5] We assume that the set $\mathrm{Succs}(s, a) = \{s' \in S \mid P(s, a, s') > 0\}$ of *relevant a-successors* of state s is returned in some arbitrary, but fixed order.

Lemma 4. *Procedure* $\mathsf{Strengthen}_{\mathcal{H}}$ *is sound, i.e., it satisfies the specification in Definition 4.*

Theorem 3. *Procedure* $\mathsf{PrIC3}_{\mathcal{H}}$ *is sound, i.e., satisfies the specification in Theorem 1.*

We remark that, analogously to standard IC3, resolving an obligation in l. 11 may be accompanied by *generalization*. That is, we attempt to update the values of multiple states at once. Generalization is, however, highly non-trivial in a probabilistic setting. We discuss three possible approaches to generalization in Sect. 6.2.

$\mathsf{Strengthen}_{\mathcal{H}}$ Terminates. We now show that $\mathsf{Strengthen}_{\mathcal{H}}$ as in Algorithm 2 terminates. The only scenario in which $\mathsf{Strengthen}_{\mathcal{H}}$ may not terminate is if it keeps spawning obligations in l. 9. Let us thus look closer at how obligations are spawned: Whenever we detect that resolving an obligation (i, s, δ) would violate relative inductivity for some action a (l. 6), we first need to update the values of the successor states $s_1, \ldots, s_n \in \mathrm{Succs}(s, a)$ in frame $i-1$, i.e., we push the obligations $(i-1, s_1, \delta_1), \ldots, (i-1, s_n, \delta_n)$ which have to be resolved first (ll. 7–9). It is noteworthy that, for a TS, a single action leads to a single successor state s_1. Algorithm 2 employs a heuristic $\mathcal{H}$ to determine the probabilities required for pushing obligations (l. 7). Assume for an obligation (i, s, δ) that the check in l. 6 yields $\exists a \in \mathrm{Act}(s) : \Phi_a(F_{i-1})[s] > \delta$. Then $\mathcal{H}$ takes s, a, δ and reports some probability δ_j for every a-successor s_j of s. However, an arbitrary heuristic of type $\mathcal{H}\colon S \times \mathrm{Act} \times [0,1] \to [0,1]^*$ may lead to non-terminating behavior: If $\delta_1, \ldots, \delta_n = F_{i-1}[s_1], \ldots F_{i-1}[s_n]$, then the heuristic has no effect. It is thus natural to require that an *adequate* heuristic $\mathcal{H}$ yields probabilities such that the check $\Phi_a(F_{i-1})[s] > \delta$ in l. 6 cannot succeed twice for the *same obligation* (i, s, δ) and *same action* a. Formally, this is guaranteed by the following:

Definition 5. *Heuristic* $\mathcal{H}$ *is* adequate *if the following triple is valid (for any frame* F*):*

$$\begin{array}{l} \{\ \mathit{Succs}(s,a) = s_1, \ldots, s_n\ \} \\ \quad \delta_1, \ldots, \delta_n \ \leftarrow\ \mathcal{H}(s, a, \delta) \\ \{\ \Phi_a\left(F\,\langle s_1 \mapsto \delta_1\rangle \ldots \langle s_n \mapsto \delta_n\rangle\right)[s] \le \delta\ \} \end{array}$$ △

Details regarding our implementation of heuristic $\mathcal{H}$ are found in Sect. 6.1.

For an adequate heuristic, attempting to resolve an obligation (i, s, δ) (ll. 3 – 11) either succeeds after spawning it at most $|\mathrm{Act}(s)|$ times or $\mathsf{Strengthen}_{\mathcal{H}}$ returns `false`. By a similar argument, attempting to resolve an obligation $(i > 0, s, \_\_)$ leads to at most $\sum_{a \in \mathrm{Act}(s)} |\{s' \in S \mid P(s, a, s') > 0\}|$ other obligations of the form $(i-1, s', \_\_)$. Consequently, the total number of obligations spawned by Algorithm 2 is bounded. Since Algorithm 2 terminates if all obligations have been resolved (l. 12) and each of its loop iterations either returns `false`, spawns obligations, or resolves an obligation, we conclude:

Lemma 5. $\mathsf{Strengthen}_{\mathcal{H}}(F_0, \ldots, F_k)$ *terminates for every adequate heuristic* $\mathcal{H}$.

Recovery Statement 3. *Let* $\mathcal{H}$ *be adequate. Then for qualitative reachability* ($\lambda = 0$)*, all obligations spawned by* $\mathsf{Strengthen}_{\mathcal{H}}$ *as in Algorithm 2 are of the form* $(i, s, 0)$.

$\mathsf{Strengthen}_{\mathcal{H}}$ returns `false`. There are two cases in which $\mathsf{Strengthen}_{\mathcal{H}}$ fails to restore the PrIC3 invariants and returns `false`. The first case (the left disjunct of l. 4) is that we encounter an obligation for frame F_0. Resolving such an obligation would inevitably violate *initiality*; analogously to standard IC3, we thus return `false`.

The second case (the right disjunct of l. 4) is that we encounter an obligation (i, s, δ) for a bad state $s \in B$ with a probability $\delta < 1$ (though, obviously, all $s \in B$ have probability $=1$). Resolving such an obligation would inevitably prevents us from restoring *relative inductivity*: If we updated $F_i[s]$ to δ, we would have $\Phi(F_{i-1})[s] = 1 > \delta = F_i[s]$. Notice that, in contrast to standard IC3, this second case *can* occur in PrIC3:

Example 4. Assume we have to resolve an obligation $(i, s_3, 1/2)$ for the MDP in Fig. 1. This involves spawning obligations $(i{-}1, s_4, \delta_1)$ and $(i{-}1, s_5, \delta_2)$, where s_5 is a bad state, such that $1/3 \cdot \delta_1 + 2/3 \cdot \delta_2 \leq 1/2$. Even for $\delta_1 = 0$, this is only possible if $\delta_2 \leq 3/4 < 1$. △

$\mathsf{Strengthen}_{\mathcal{H}}$ Cannot Prove Unsafety. If standard IC3 returns `false`, it proves unsafety by constructing a counterexample, i.e., *a single path from the initial state to a bad state*. If PrIC3 returns `false`, there are two possible reasons: *Either* the MDP is indeed unsafe, *or* the heuristic $\mathcal{H}$ at some point selected probabilities in a way such that $\mathsf{Strengthen}_{\mathcal{H}}$ is unable to restore the PrIC3 invariants (even though the MDP might in fact be safe). $\mathsf{Strengthen}_{\mathcal{H}}$ thus only returns a *potential* counterexample which either proves unsafety or indicates that our heuristic was inappropriate.

Counterexamples in our case consist of subsystems rather than a single path (see Challenge 2 and Sect. 5). $\mathsf{Strengthen}_{\mathcal{H}}$ hence returns the set Q.touched() of all states that eventually appeared in the obligation queue. This set is a conservative approximation, and optimizations as in [1] may be beneficial. Furthermore, in the qualitative case, our potential counterexample subsumes the counterexamples constructed by standard IC3:

Recovery Statement 4. *Let* $\mathcal{H}_0$ *be the adequate heuristic mapping every state to* 0*. For qual. reachability* ($\lambda = 0$)*, if success =* `false` *is returned by* $\mathsf{Strengthen}_{\mathcal{H}_0}(F_0, \ldots, F_k)$*, then* Q*.touched*() *contains a path from the initial to a bad state.*[6]

[6] Q.touched() might be restricted to only contain this path by some simple adaptions.

Data: global MDP $\mathfrak{M}$, set of bad states B, threshold λ
Result: `true` iff $\Pr^{\max}(s_I \models \Diamond B) \leq \lambda$

```
Ω ← Initialize(); touched ← {s_I};
do
    H ← CreateHeuristic(Ω);   safe, subsystem ← PrIC3_H();
    if safe then return true ;
    if CheckRefutation(subsystem) then return false ;
    touched ← Enlarge(touched, subsystem);
    Ω ← Refine(Ω, touched);
while touched ≠ S;
return Ω(s_I) ≤ λ
```

Algorithm 3: PrIC3: The outermost loop dealing with possibly imprecise heuristics

5 Dealing with Potential Counterexamples

Recall that our core algorithm PrIC3$_{\mathcal{H}}$ is incomplete for a fixed heuristic $\mathcal{H}$: It cannot give a conclusive answer whenever it finds a potential counterexample for two possible reasons: Either the heuristic $\mathcal{H}$ turned out to be inappropriate or the MDP is indeed unsafe. The idea to overcome the former is to call PrIC3$_{\mathcal{H}}$ finitely often in an outer loop that generates new heuristics until we find an appropriate one: If PrIC3$_{\mathcal{H}}$ still does not report safety of the MDP, then it is indeed unsafe. We do not blindly generate new heuristics, but use the potential counterexamples returned by PrIC3$_{\mathcal{H}}$ to refine the previous one.

Let consider the procedure PrIC3 in Algorithm 3 which wraps our core algorithm PrIC3$_{\mathcal{H}}$ in more detail: First, we create an *oracle* $\Omega\colon S \to [0,1]$ which (roughly) *estimates* the probability of reaching B for every state. A *perfect oracle* would yield *precise* maximal reachability probabilites, i.e., $\Omega(s) = \Pr^{\max}(s \models \Diamond B)$ for every state s. We construct oracles by user-supplied methods (highlighted in blue). Examples of implementations of all user-supplied methods in Algorithm 3 are discussed in Sect. 7.

Assuming the oracle is good, but not perfect, we construct an adequate heuristic $\mathcal{H}$ selecting probabilities based on the oracle[7] for all successors of a given state: There are various options. The simplest is to pass-through the oracle values. A version that is more robust against noise in the oracle is discussed in Sect. 6. We then invoke PrIC3$_{\mathcal{H}}$. If PrIC3$_{\mathcal{H}}$ reports safety, the MDP is indeed safe by the soundness of PrIC3$_{\mathcal{H}}$.

Check Refutation. If PrIC3$_{\mathcal{H}}$ does not report safety, it reports a subsystem that hints to a *potential* counterexample. Formally, this subsystem is a subMDP of states that were 'visited' during the invocation of Strengthen$_{\mathcal{H}}$.

Definition 6 (subMDP). *Let $\mathfrak{M} = (S, s_I, \mathrm{Act}, P)$ be an MDP and let $S' \subseteq S$ with $s_I \in S'$. We call $\mathfrak{M}_{S'} = (S', s_I, \mathrm{Act}, P')$ the* subMDP induced by $\mathfrak{M}$ and S', *where for all $s, s' \in S'$ and all $a \in \mathrm{Act}$, we have $P'(s,a,s') = P(s,a,s')$.* △

[7] We thus assume that heuristic $\mathcal{H}$ invokes the oracle whenever it needs to guess some probability.

A subMDP $\mathfrak{M}_{S'}$ may be substochastic where missing probability mass never reaches a bad state. Definition 1 is thus relaxed: For all states $s \in S'$ we require that $\sum_{s' \in S'} P(s,a,s') \leq 1$.If the subsystem is unsafe, we can conclude that the original MDP $\mathfrak{M}$ is also safe.

Lemma 6. *If $\mathfrak{M}'$ is a subMDP of $\mathfrak{M}$ and $\mathfrak{M}'$ is unsafe, then $\mathfrak{M}$ is also unsafe.*

The role of CheckRefutation is to establish whether the subsystem is indeed a true counterexample or a spurious one. Formally, CheckRefutation should ensure:

$$\{\,\texttt{true}\,\}\ \mathit{res} \leftarrow \mathsf{CheckRefutation}\,(\mathit{subsystem})\ \{\,\mathit{res} = \texttt{true} \ \Leftrightarrow\ \mathfrak{M}_{\mathit{subsystem}} \text{ unsafe}\,\}.$$

Again, PrIC3 is backward compatible in the sense that a single fixed heuristic is always sufficient when reasoning about reachability ($\lambda = 0$).

Recovery Statement 5. *For qualitative reachability ($\lambda = 0$) and the heuristic $\mathcal{H}_0$ from Recovery Statement 4,* PrIC3 *invokes its core* PrIC3$_{\mathcal{H}}$ *exactly* once.

This statement is true, as PrIC3$_{\mathcal{H}}$ returns either *safe* or a subsystem containing a path from the initial state to a bad state. In the latter case, CheckRefutation detects that the subsystem is indeed a counterexample which cannot be spurious in the qualitative setting.

We remark that the procedure CheckRefutation invoked in l. 5 is a classical fallback; it runs an (alternative) model checking algorithm, e.g., solving the set of Bellman equations, for the subsystem. In the worst case, i.e., for $S' = S$, we thus solve exactly our problem statement. Empirically (Table 1) we observe that for reasonable oracles the procedure CheckRefutation is invoked on significantly smaller subMDPs. However, in the worst case the subMDP must include *all* paths of the original MDP, and then thus coincides.

Refine Oracle. Whenever we have neither proven the MDP safe nor unsafe, we refine the oracle to prevent generating the same subsystem in the next invocation of PrIC3$_{\mathcal{H}}$. To ensure termination, oracles should only be refined finitely often. That is, we need some progress measure. The set *touched* overapproximates all counterexamples encountered in some invocation of PrIC3$_{\mathcal{H}}$ and we propose to use its size as the progress measure. While there are several possibilities to update *touched* through the user-defined procedure Enlarge (l. 6), every implementation should hence satisfy $\{\,\texttt{true}\,\}\ \mathit{touched}' \leftarrow \mathsf{Enlarge}(\mathit{touched}, \_\_)\ \{\,|\mathit{touched}'| > |\mathit{touched}|\,\}$. Consequently, after finitely many iterations, the oracle is refined with respect to all states. In this case, we may as well rely on solving the characteristic LP problem:

Lemma 7. *The algorithm* PrIC3 *in Algorithm 3 is sound and complete if* Refine(Ω, S) *returns a perfect oracle Ω (with S is the set of all states).*

Weaker assumptions on Refine are possible, but are beyond the scope of this paper. Moreover, the above lemma does not rely on the abstract concept that heuristic $\mathcal{H}$ provides suitable probabilities after finitely many refinements.[8]

[8] One could of course now also create a heuristic that is trivial for a perfect oracle and invoke PrIC3$_{\mathcal{H}}$ with the heuristic for the perfect oracle, but there really is no benefit in doing so.

6 Practical PrIC3

So far, we gave a conceptual view on PrIC3, but now take a more practical stance. We detail important features of effective implementations of PrIC3 (based on our empirical evaluation). We first describe an implementation without generalization, and then provide a prototypical extension that allows for three variants of generalization.

6.1 A Concrete PrIC3 Instance Without Generalization

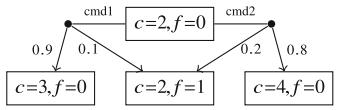
Input. We describe MDPs using the Prism guarded command language[9], exemplified in Fig. 3. States are described by valuations to m (integer-valued) program variables vars, and outgoing actions are described by commands of the form

```
[] guard -> prob1 : update1 & ... & probk : updatek
```

If a state satisfies `guard`, then the corresponding action with k branches exists; probabilities are given by `probi`, the successor states are described by `updatei`, see Fig. 3b.

```
module ex
  c : [0..20] init 0;    f : [0..1] init 0;
  [] c<20 -> 0.1:(f'=1) + 0.9:(c'=c+1); // cmd 1
  [] c<10 -> 0.2:(f'=1) + 0.8:(c'=c+2);
endmodule
```

(a) Prism code snippet



(b) Part of the corresponding MDP

Fig. 3. Illustrative Prism-style probabilistic guarded command language example

Encoding. We encode frames as logical formulae. Updating frames then corresponds to adding conjuncts, and checking for relative inductivity is a satisfiability call. Our encoding is as follows: States are assignments to the program variables, i.e., $\mathsf{States} = \mathbb{Z}^m$. We use various uninterpreted functions, to whom we give semantics using appropriate constraints. Frames[10] are represented by uninterpreted functions $\mathsf{Frame}\colon \mathsf{States} \to \mathbb{R}$ satisfying $\mathsf{Frame}(s) = d$ implies $F[s] \geq d$. Likewise, the Bellman operator is an uninterpreted function $\mathsf{Phi}\colon \mathsf{States} \to \mathbb{R}$ such that $\mathsf{Phi}(s) = d$ implies $\Phi(F)[s] \geq d$. Finally, we use $\mathsf{Bad}\colon \mathsf{States} \to \mathbb{B}$ with $\mathsf{Bad}(s)$ iff $s \in B$.

Among the appropriate constraints, we ensure that variables are within their range, bound the values for the frames, and enforce $\mathsf{Phi}(s) = 1$ for $s \in B$. We encode the guarded commands as exemplified by this encoding of the first command in Fig. 3:

$$\forall s \in \mathsf{States}\colon \neg\mathsf{Bad}(s) \wedge s[c] < 20 \implies \mathsf{Phi}(s) = 0.1 \cdot \mathsf{Frame}((s[c], 1)) + 0.9 \cdot \mathsf{Frame}((s[c]+1, s[f]))\,.$$

[9] Preprocessing ensures a single thread (module) and no deadlocks.
[10] In each operation, we only consider a single frame.

In our implementation, we optimize the encoding. We avoid the uninterpreted functions by applying an adapted Ackerman reduction. We avoid universal quantifiers, by first observing that we always ask whether a single state is not inductive, and then unfolding the guarded commands in the constraints that describe a frame. That encoding grows linear in the size of the maximal out-degree of the MDP, and is in the quantifier-free fragment of linear arithmetic (QFLRIA).

Heuristic. We select probabilities δ_i by solving the following optimization problem, with variables x_i, $\mathit{range}(x_i) \in [0,1]$, for states $s_i \in \mathrm{Succs}(s,a)$ and oracle Ω[11].

$$\text{minimize} \sum_{\substack{i \\ s_i \notin B}}^{k} \left| \frac{x_i}{\sum_{j=1}^{k} x_j} - \frac{\Omega(s_i)}{\sum_{j=1}^{n} \Omega(s_j)} \right| \text{ s.t. } \delta = \sum_{i=1}^{k} P(s,a,s_i) \cdot \begin{cases} 1, & \text{if } s_i \in B, \\ x_i, & \text{else.} \end{cases}$$

The constraint ensures that, if the values x_i correspond to the actual reachability probabilities from s_i, then the reachability from state s is exactly δ. A constraint stating that $\delta \geq \ldots$ would also be sound, but we choose equality as it preserves room between the actual probability and the threshold we want to show. Finally, the objective function aims to preserve the ratio between the suggested probabilities.

Repushing and Breaking Cycles. Repushing [23] is an essential ingredient of both standard IC3 and PrIC3. Intuitively, we avoid opening new frames and spawning obligations that can be deduced from current information. Since repushing generates further obligations in the current frame, its implementation requires that the detection of Zeno-behavior has to be moved from $\mathsf{PrIC3}_{\mathcal{H}}$ into the $\mathsf{Strengthen}_{\mathcal{H}}$ procedure. Therefore, we track the histories of the obligations in the queue. Furthermore, once we detect a cycle we first try to adapt the heuristic $\mathcal{H}$ locally to overcome this cyclic behavior instead of immediately giving up. This local adaption reduces the number of $\mathsf{PrIC3}_{\mathcal{H}}$ invocations.

Extended Queue. In contrast to standard IC3, the obligation queue might contain entries that vary only in their δ entry. In particular, if the MDP is not a tree, it may occur that the queue contains both (i,s,δ) and (i,s,δ') with $\delta > \delta'$. Then, (i,s,δ') can be safely pruned from the queue. Similarly, after handling (i,s,δ), if some fresh obligation $(i,s,\delta'' > \delta)$ is pushed to the queue, it can be substituted with (i,s,δ). To efficiently operationalize these observations, we keep an additional mapping which remains intact over multiple invocations of $\mathsf{Strengthen}_{\mathcal{H}}$. We furthermore employed some optimizations for Q.touched() aiming to track potential counterexamples better. After refining the heuristic, one may want to reuse frames or the obligation queue, but empirically this leads to performance degradation as the values in the frames are inconsistent with behavior suggested by the heuristic.

[11] If $\max \Omega(s_j) = 0$, we assume $\forall j.\Omega(s_j) = 0.5$. If $\delta = 0$, we omit rescaling to allow $\sum x_j = 0$.

6.2 Concrete PrIC3 with Generalization

So far, frames are updated by changing single entries whenever we resolve obligations (i, s, δ), i.e., we add conjunctions of the form $F_i[s] \leq \delta$. Equivalently, we may add a constraint $\forall s' \in S : F_i[s'] \leq p_{\{s\}}(s')$ with $p_{\{s\}}(s) = \delta$ and $p_{\{s\}} = 1$ for all $s' \neq s$.

Generalization in IC3 aims to update a set G (including s) of states in a frame rather than a single one without invalidating relative inductivity. In our setting, we thus consider a function $p_G \colon G \to [0, 1]$ with $p_G(s) \leq \delta$ that assigns (possibly different) probabilities to all states in G. Updating a frame then amounts to adding the constraint

$$\forall\, s \in \mathsf{States} \colon\ s \in G \implies \mathsf{Frame}\,(s) \leq p_G(s).$$

Standard IC3 generalizes by iteratively "dropping" a variable, say v. The set G then consists of all states that do not differ from the fixed state s except for the value of v.[12] We take the same approach by iteratively dropping program variables. Hence, p_G effectively becomes a mapping from the value $s[v]$ to a probability. We experimented with four types of functions p_G that we describe for Markov chains. The ideas are briefly outlined below; details are beyond the scope of this paper.

Constant p_G. Setting all $s \in G$ to δ is straightforward but empirically not helpful.

Linear Interpolation. We use a linear function p_G that interpolates two points. The first point $(s[v], \delta)$ is obtained from the obligation (i, s, δ). For a second point, consider the following: Let Com be the unique[13] command active at state s. Among all states in G that are enabled in the guard of Com, we take the state s' in which $s'[v]$ is maximal[14]. The second point for interpolation is then $(s'[v], \Phi\,(F_{i-1})\,[s'])$. If the relative inductivity fails for p_G we do not generalize with p_G, but may attempt to find other functions.

Polynomial Interpolation. Rather than linearly interpolating between two points, we may interpolate using more than two points. In order to properly fit these points, we can use a higher-degree polynomial. We select these points using counterexamples to generalization (CTGs): We start as above with linear interpolation. However, if p_G is not relative inductive, the SMT solver yields a model with state $s'' \in G$ and probability δ'', with s'' violating relative inductivity, i.e., $\Phi\,(F_{i-1})\,[s''] > \delta''$. We call $(s'', \Phi\,(F_{i-1})\,[s''])$ a CTG, and $(s''[v], \Phi\,(F_{i-1})\,[s'']))$ is then a further interpolation point, and we repeat.

Technically, when generalizing using nonlinear constraints, we use real-valued arithmetic with a branch-and-bound-style approach to ensure integer values.

[12] Formally, $G = \{s' \mid \text{for all } v' \in \text{vars} \setminus \{v\} : s'(v') = s(v')\}$.
[13] Recall that we have a Markov chain consisting of a single module.
[14] This implicitly assumes that v is increased. Adaptions are possible.

Hybrid Interpolation. In polynomial interpolation, we generate high-degree polynomials and add them to the encoding of the frame. In subsequent invocations, reasoning efficiency is drastically harmed by these high-degree polynomials. Instead, we soundly approximate p_G by a piecewise linear function, and use these constraints in the frame.

7 Experiments

We assess how PrIC3 may contribute to the state of the art in probabilistic model checking. We do some early empirical evaluation showing that PrIC3 is feasible. We see ample room for further improvements of the prototype.

Implementation. We implemented a prototype[15] of PrIC3 based on Sect. 6.1 in Python. The input is represented using efficient data structures provided by the model checker Storm. We use an incremental instance of Z3 [47] for each frame, as suggested in [23]. A solver for each frame is important to reduce the time spent on pushing the large frame-encodings. The optimization problem in the heuristic is also solved using Z3. All previously discussed generalizations (none, linear, polynomial, hybrid) are supported.

Oracle and Refinement. We support the (pre)computation of four different types of oracles for the initialization step in Algorithm 3: (1) A perfect oracle solving *exactly* the Bellman equations. Such an oracle is unrealistic, but interesting from a conceptual point. (2) Relative frequencies by recording all visited states during simulation. This idea is a naïve simplification of Q-learning. (3) Model checking with decision diagrams (DDs) and few value iterations. Often, a DD representation of a model can be computed fast, and the challenge is in executing sufficient value iterations. We investigate whether doing few value iterations yields a valuable oracle (and covers states close to bad states). (4) Solving a (pessimistic) LP from BFS partial exploration. States that are not expanded are assumed bad. Roughly, this yields oracles covering states close to the initial states.

To implement Refine (cf. Algorithm 3, l. 7), we create an LP for the subMDP induced by the touched states. For states whose successors are not in the touched states, we add a transition to B labeled with the oracle value as probability. The solution of the resulting LP updates the entries corresponding to the touched states.

For Enlarge (cf. Algorithm 3, l. 6), we take the union of the subsystem and the touched states. If this does not change the set of touched states, we also add its successors.

Setup. We evaluate the run time and memory consumption of our prototype of PrIC3. We choose a combination of models from the literature (BRP [21], ZeroConf [18]) and some structurally straightforward variants of grids (chain, double chain; see [11, Appendix A]). Since our prototype lacks the sophisticated

[15] The prototype is available open-source from https://github.com/moves-rwth/PrIC3.

preprocessing applied by many state-of-the-art model checkers, it is more sensitive to the precise encoding of a model, e.g., the number of commands. To account for this, we generated new encodings for all models. All experiments were conducted on an single core of an Intel® Xeon® Platinum 8160 processor. We use a 15 min time-limit and report TO otherwise. Memory is limited to 8GB; we report MO if it is exceeded. Apart from the oracle, all parameters of our prototype remain fixed over all experiments. To give an impression of the run times, we compare our prototype with both the explicit ($\text{Storm}_{\text{sparse}}$) and DD-based ($\text{Storm}_{\text{dd}}$) engine of the model checker Storm 1.4, which compared favourably in QComp [29].

Results. In Table 1, we present the run times for various invocations of our prototype and Oracle 4^{16}. In particular, we give the model name and the number of (non-trivial) states in the particular instance, and the (estimated) actual probability to reach B. For each model, we consider multiple thresholds λ. The next 8 columns report on the four variants of PrIC3 with varying generalization schemes. Besides the scheme with the run times, we report for each scheme the number of states of the largest (last) subsystem that CheckRefutation in Algorithm 3, l. 5 was invoked upon (column $|sub|$). The last two columns report on the run times for Storm that we provide for comparison. In each row, we mark with purple MDPs that are unsafe, i.e., PrIC3 refutes these MDPs for the given threshold λ. We **highlight** the best configurations of PrIC3.

Discussion. Our experiments give a mixed picture on the performance of our implementation of PrIC3. On the one hand, Storm significantly outperforms PrIC3 on most models. On the other hand, PrIC3 is capable of reasoning about huge, yet simple, models with up to 10^{12} states that Storm is unable to analyze within the time and memory limits. There is more empirical evidence that PrIC3 may complement the state-of-the-art:

First, *the size of thresholds matters*. Our benchmarks show that—at least without generalization—more "wiggle room" between the precise maximal reachability probability and the threshold generally leads to a better performance. PrIC3 may thus prove bounds for large models where a precise quantitative reachability analysis is out of scope.

Second, PrIC3 *enjoys the benefits of bounded model checking*. In some cases, e.g., ZeroConf for $\lambda = 0.45$, PrIC3 refutes very fast as it does not need to build the whole model.

Third, if PrIC3 proves the safety of the system, it does so without relying on checking large subsystems in the CheckRefutation step.

Fourth, *generalization is crucial*. Without generalization, PrIC3 is unable to prove safety for any of the considered models with more than 10^3 states. With generalization, however, it can prove safety for very large systems and thresholds close to the exact reachability probability. For example, it proved safety of the

[16] We explore $\min\{|S|, 5000\}$ states using BFS and Storm.

Table 1. Empirical results. Run times are in seconds; time out = 15 min.

	$\lvert S\rvert$	$\Pr^{\max}(s_I \models \Diamond B)$	λ	w/o	$\lvert sub\rvert$	lin	$\lvert sub\rvert$	pol	$\lvert sub\rvert$	hyb	$\lvert sub\rvert$	$\text{Storm}_{\text{sparse}}$	Storm_{dd}
BRP	10^3	0.035	0.1	TO	–	TO	–	TO	–	TO	–	<0.1	0.12
			0.01	**51.3**	324	125.8	324	TO	–	MO	–	<0.1	0.18
			0.005	**10.9**	188	38.3	188	TO	–	MO	–	<0.1	0.1
ZeroConf	10^4	0.5	0.9	TO	–	TO	–	0.4	0	**0.1**	0	<0.1	296.8
			0.52	TO	–	TO	–	0.2	0	**0.2**	0	<0.1	282.6
			0.45	**<0.1**	1	**<0.1**	1	**<0.1**	1	**<0.1**	1	<0.1	300.2
	10^9	∼0.55	0.9	TO	–	TO	–	**3.7**	0	MO	–	MO	TO
			0.75	TO	–	TO	–	**3.4**	0	MO	–	MO	TO
			0.52	TO	–	TO	–	TO	–	TO	–	MO	TO
			0.45	**<0.1**	1	**<0.1**	1	**<0.1**	1	**<0.1**	1	MO	TO
Chain	10^3	0.394	0.9	18.8	0	60.2	0	1.2	0	**0.3**	0	<0.1	<0.1
			0.4	20.1	0	55.4	0	**0.9**	0	TO	–	<0.1	<0.1
			0.35	**91.8**	431	119.5	431	TO	–	TO	–	<0.1	<0.1
			0.3	**46.1**	357	64.0	357	TO	–	TO	–	<0.1	<0.1
	10^4	0.394	0.9	TO	–	TO	–	1.6	0	**0.3**	0	<0.1	4.5
			0.4	TO	–	TO	–	**1.4**	0	TO	–	<0.1	4.9
			0.3	TO	–	TO	–	TO	–	TO	–	<0.1	4.9
	10^{12}	0.394	0.9	TO	–	TO	–	**6.4**	0	MO	–	MO	TO
			0.4	TO	–	TO	–	**6.0**	0	MO	–	MO	TO
Double chain	10^3	0.215	0.9	528.1	0	828.8	0	203.3	0	**0.6**	0	<0.1	<0.1
			0.3	588.4	0	TO	–	138.3	0	**0.5**	0	<0.1	<0.1
			0.216	**597.4**	0	TO	–	765.8	0	MO	–	<0.1	<0.1
			0.15	TO	–	TO	–	TO	–	TO	–	<0.1	<0.1
	10^4	0.22	0.3	TO	–	TO	–	17.5	0	**0.5**	0	0.2	2.6
			0.24	TO	–	TO	–	**16.8**	0	MO	–	0.2	2.7
	10^7	$2.6\mathbf{E}^{-4}$	$4\mathbf{E}^{-3}$	TO	–	TO	–	TO	–	MO	–	TO	TO
			$2.7\mathbf{E}^{-4}$	TO	–	TO	–	**281.2**	0	MO	–	TO	TO

Chain benchmark with 10^{12} states for a threshold of 0.4 which differs from the exact reachability probability by 0.006.

Fifth, *there is no best generalization*. There is no clear winner out of the considered generalization approaches. Linear generalization always performs worse than the other ones. In fact, it performs worse than no generalization at all. The hybrid approach, however, occasionally has the edge over the polynomial approach. This indicates that more research is required to find suitable generalizations.

In [11, Appendix A], we also compare the additional three types of oracles (1–3). We observed that only few oracle refinements are needed to prove *safety*; for small models at most one refinement was sufficient. However, this does not hold if the given MDP is unsafe. DoubleChain with $\lambda = 0.15$, for example, and Oracle 2 requires 25 refinements.

8 Conclusion

We have presented PrIC3—the first truly probabilistic, yet conservative, extension of IC3 to quantitative reachability in MDPs. Our theoretical development

is accompanied by a prototypical implementation and experiments. We believe there is ample space for improvements including an in-depth investigation of suitable oracles and generalizations.

References

1. Ábrahám, E., Becker, B., Dehnert, C., Jansen, N., Katoen, J.-P., Wimmer, R.: Counterexample generation for discrete-time Markov models: an introductory survey. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E.B., Schaefer, I. (eds.) SFM 2014. LNCS, vol. 8483, pp. 65–121. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07317-0_3
2. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018)
3. Agrawal, S., Chatterjee, K., Novotný, P.: Lexicographic ranking supermartingales: an efficient approach to termination of probabilistic programs. In: PACMPL 2(POPL), pp. 34:1–34:32 (2018)
4. 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
5. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. Handbook of Model Checking, pp. 963–999. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
6. Baier, C., Hermanns, H., Katoen, J.-P.: The 10,000 facets of MDP model checking. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 420–451. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_21
7. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
8. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: interval iteration for markov decision processes. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 160–180. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_8
9. Barthe, G., Espitau, T., Ferrer Fioriti, L.M., Hsu, J.: Synthesizing probabilistic invariants via Doob's decomposition. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 43–61. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_3
10. Bartocci, E., Kovács, L., Stankovič, M.: Automatic generation of moment-based invariants for prob-solvable loops. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 255–276. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_15
11. Batz, K., Junges, S., Kaminski, B.L., Katoen, J.-P., Matheja, C., Schröer, P.: Pric3: Property directed reachability for MDPS. ArXiv e-prints (2020). https://arxiv.org/abs/2004.14835
12. Biere, A.: Bounded model checking, Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press (2009)
13. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7

14. Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínský, J., Kwiatkowska, M., Parker, D., Ujma, M.: 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
15. Chadha, R., Viswanathan, M.: A counterexample-guided abstraction-refinement framework for Markov decision processes. ACM Trans. Comput. Logist. **12**(1), 1:1–1:49 (2010)
16. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
17. Chakraborty, S., Fried, D., Meel, K.S., Vardi, M.Y.: From weighted to unweighted model counting. In: IJCAI, pp. 689–695. AAAI Press (2015)
18. Cheshire, S., Aboba, B., Guttman, E.: Dynamic configuration of ipv4 link-local addresses. RFC **3927**, 1–33 (2005)
19. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. FMSD **49**(3), 190–218 (2016)
20. D'Argenio, P.R., Hartmanns, A., Sedwards, S.: Lightweight statistical model checking in nondeterministic continuous time. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 336–353. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_22
21. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: de Alfaro, L., Gilmore, S. (eds.) PAPM-PROBMIV 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44804-7_3
22. 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
23. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD, pp. 125–134. FMCAD Inc. (2011)
24. 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
25. Gretz, F., Katoen, J.-P., McIver, A.: Prinsys—On a Quest for Probabilistic Loop Invariants. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 193–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_17
26. 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)
27. Gurfinkel, A., Ivrii, A.: Pushing to the top. In: FMCAD, pp. 65–72. IEEE (2015)
28. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018)
29. Hahn, E.M., Hartmanns, A., Hensel, C., Klauck, M., Klein, J., Křetínský, J., Parker, D., Quatmann, T., Ruijters, E., Steinmetz, M.: The 2019 comparison of tools for the analysis of quantitative formal models. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 69–92. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_5

30. 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
31. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: a web-based probabilistic model checker. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 312–317. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_22
32. Han, T., Katoen, J.-P., Damman, B.: Counterexample generation in probabilistic model checking. IEEE Trans. Software Eng. **35**(2), 241–257 (2009)
33. Hark, M., Kaminski, B.L., Giesl, J., Katoen, J.-P.: Aiming low is harder: Induction for lower bounds in probabilistic program verification. In: PACMPL 4(POPL), 37:1–37:28 (2020)
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., Kaminski, B.L.: Optimistic value iteration. CAV. LNCS, Springer (2020). [to appear]
36. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: FMCAD, pp. 157–164. IEEE (2013)
37. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_16
38. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
39. Kaminski, B.L.: Advanced Weakest Precondition Calculi for Probabilistic Programs. Ph.D. thesis, RWTH Aachen University, Germany (2019). http://publications.rwth-aachen.de/record/755408/files/755408.pdf
40. Kaminski, B.L., Katoen, J.-P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected runtimes of randomized algorithms. J. ACM **65**(5), 30:1–30:68 (2018)
41. 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)
42. Kozen, D.: A probabilistic PDL. In: STOC, pp. 291–297. ACM (1983)
43. 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
44. Lange, T., Neuhäußer, M.R., Noll, T., Katoen, J.-P.: IC3 software model checking. In: STTT, vol. 22, pp. 135–161 (2020)
45. Lassez, J.L., Nguyen, V.L., Sonenberg, L.: Fixed point theorems and semantics: a folk tale. Inf. Process. Lett. **14**(3), 112–116 (1982)
46. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer, New York (2005). https://doi.org/10.1007/b138392
47. 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

48. Park, D.: Fixpoint induction and proofs of program properties. Machine intelligence **5**, 59–78 (1969)
49. Polgreen, E., Brain, M., Fränzle, M., Abate, A.: Verifying reachability properties in Markov chains via incremental induction. CoRR abs/1909.08017 (2019)
50. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley, Hoboken (1994)
51. Quatmann, T., Katoen, J.-P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 643–661. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_37
52. 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
53. Seufert, T., Scholl, C.: Sequential verification using reverse PDR. MBMV. pp. 79–90. Shaker Verlag (2017)
54. Suenaga, K., Ishizawa, T.: Generalized property-directed reachability for hybrid systems. In: Beyer, D., Zufferey, D. (eds.) VMCAI 2020. LNCS, vol. 11990, pp. 293–313. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-39322-9_14
55. Takisaka, T., Oyabu, Y., Urabe, N., Hasuo, I.: Ranking and repulsing supermartingales for reachability in probabilistic programs. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 476–493. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_28
56. Vazquez-Chanlatte, M., Rabe, M.N., Seshia, S.A.: A model counter's guide to probabilistic systems. CoRR abs/1903.09354 (2019)
57. 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

Synthesis

Good-Enough Synthesis

Shaull Almagor[1(✉)] and Orna Kupferman[2]

[1] Department of Computer Science, Technion, Haifa, Israel
shaull@cs.technion.ac.il
[2] School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel
orna@cs.huji.ac.il

Abstract. We introduce and study *good-enough synthesis* (GE-synthesis) – a variant of synthesis in which the system is required to satisfy a given specification ψ only when it interacts with an environments for which a satisfying interaction exists. Formally, an input sequence x is *hopeful* if there exists some output sequence y such that the induced computation $x \otimes y$ satisfies ψ, and a system GE-realizes ψ if it generates a computation that satisfies ψ on all hopeful input sequences. GE-synthesis is particularly relevant when the notion of correctness is *multi-valued* (rather than Boolean), and thus we seek systems of the highest possible quality, and when synthesizing *autonomous systems*, which interact with unexpected environments and are often only expected to do their best.

We study GE-synthesis in Boolean and multi-valued settings. In both, we suggest and solve various definitions of GE-synthesis, corresponding to different ways a designer may want to take hopefulness into account. We show that in all variants, GE-synthesis is not computationally harder than traditional synthesis, and can be implemented on top of existing tools. Our algorithms are based on careful combinations of nondeterministic and universal automata. We augment systems that GE-realize their specifications by monitors that provide satisfaction information. In the multi-valued setting, we provide both a worst-case analysis and an expectation-based one, the latter corresponding to an interaction with a stochastic environment.

1 Introduction

Synthesis is the automated construction of a system from its specification: given a specification ψ, typically by a linear temporal logic (LTL) formula over sets I and O of input and output signals, the goal is to construct a finite-state system that satisfies ψ [9,20]. At each moment in time, the system reads an assignment, generated by the environment, to the signals in I, and responds with an assignment to the signals in O. Thus, with every input sequence, the system associates an output sequence. The system *realizes* ψ if ψ is satisfied in all the interactions of the system, with all environments [5].

S. Almagor—Supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 837327.
O. Kupferman—Supported in part by the Israel Science Foundation, grant No. 2357/19.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 541–563, 2020.
https://doi.org/10.1007/978-3-030-53291-8_28

In practice, the requirement to satisfy the specification in all environments is often too strong. Accordingly, it is common to add assumptions on the behavior of the environment. An assumption may be direct, say given by an LTL formula that restricts the set of possible input sequences [8], less direct, say a bound on the size of the environment [13] or other resources it uses, or conceptual, say rationality from the side of the environment, which may have its own objectives [11,14]. We introduce and study a new type of relaxation of the requirement to satisfy the specification in all environments. The idea behind the relaxation is that if an environment is such that no system can interact with it in a way that satisfies the specification, then we cannot expect our system to succeed. In other words, the system has to satisfy the specification only when it interacts with environments in which this mission is possible. This is particularly relevant when synthesizing *autonomous systems*, which interact with unexpected environments and often replace human behavior, which is only expected to be *good enough* [28], and when the notion of correctness is multi-valued (rather than Boolean), and thus we seek *high-quality* systems.

Before we explain the relaxation formally, let us consider a simple example, and we start with the Boolean setting. Let $I = \{req\}$ and $O = \{grant\}$. Thus, the system receives requests and generates grants. Consider the specification $\psi = \mathsf{GF}(req \wedge grant) \wedge \mathsf{GF}(\neg req \wedge \neg grant)$. Clearly, ψ is not realizable, as an input sequence need not satisfy $\mathsf{GF}req$ or $\mathsf{GF}\neg req$. However, a system that always generates a grant upon (and only upon) a request, GE-*realizes* ψ, in the sense that for every input sequence, if there is some interaction with it with which ψ is satisfied, then our system generates such an interaction.

Formally, we model a system by a strategy $f : (2^I)^+ \to 2^O$, which given an input sequence $x = i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$, generates an output sequence $f(x) = f(i_0) \cdot f(i_0 \cdot i_1) \cdot f(i_0 \cdot i_1 \cdot i_2) \cdots \in (2^O)^\omega$, inducing the computation $x \otimes f(x) = (i_0 \cup f(i_0)) \cdot (i_i \cup f(i_0 \cdot i_1)) \cdot (i_2 \cup f(i_0 \cdot i_1 \cdot i_2)) \cdots \in (2^{I \cup O})^\omega$, obtained by "merging" x and $f(x)$. In traditional realizability, a system realizes ψ if ψ is satisfied in all environments. Formally, for all input sequences $x \in (2^I)^\omega$, the computation $x \otimes f(x)$ satisfies ψ. For our new notion, we first define when an input sequence $x \in (2^I)^\omega$ is *hopeful*, namely there is an output sequence $y \in (2^O)^\omega$ such that the computation $x \otimes y$ satisfies ψ. Then, a system GE-*realizes* ψ if ψ is satisfied in all interactions with hopeful input sequences. Formally, for all $x \in (2^I)^\omega$, if x is hopeful, then the computation $x \otimes f(x)$ satisfies ψ.

Since LTL is Boolean, synthesized systems are correct, but there is no reference to their quality. This is a crucial drawback, as designers would be willing to give up manual design only if automated-synthesis algorithms return systems of comparable quality. Addressing this challenge, researchers have developed quantitative specification formalisms. For example, in [4], the input to the synthesis problem includes also Mealy machines that grade different realizing systems. In [1], the specification formalism is the multi-valued logic LTL[$\mathcal{F}$], which augments LTL with quality operators. The satisfaction value of an LTL[$\mathcal{F}$] formula is a real value in $[0, 1]$, where the higher the value, the higher the quality in which the computation satisfies the specification. The quality operators in $\mathcal{F}$ can

prioritize and weight different scenarios. The synthesis algorithm for LTL[$\mathcal{F}$] seeks systems with a highest possible satisfaction value. One can consider either a worst-case approach, where the satisfaction value of a system is the satisfaction value of its computation with the lowest satisfaction value [1], or a stochastic approach, where it is the expected satisfaction value, given a distribution of the inputs [2].

Consider, for example, an acceleration controller of an autonomous car. Normally, the car should maintain a relatively constant speed. However, in order to optimize travel time, if a long stretch of road is visible and is identified as low-risk, the car should accelerate. Conversely, if an obstacle or some risk factor is identified, the car should decelerate. Clearly, the car cannot accelerate and decelerate at the same time. We capture this desired behavior with the following LTL[$\mathcal{F}$] formula over the inputs $\{safe, obs\}$ and outputs $\{acc, dec\}$:

$$\psi = \mathsf{G}(safe \rightarrow (acc \oplus_{\frac{2}{3}} \mathsf{X}acc)) \wedge \mathsf{G}(obs \rightarrow (dec \oplus_{\frac{3}{4}} \mathsf{X}dec)) \wedge \mathsf{G}(\neg(acc \wedge dec)).$$

Thus, in order to get satisfaction value 1, each detection of a safe stretch should be followed by an acceleration during two transactions, with a preference to the first (by the semantics of the weighted average $\oplus_\lambda$ operator, the satisfaction value of $safe \rightarrow (acc \oplus_{\frac{2}{3}} \mathsf{X}acc)$ is 1 when *safe* is followed by two *acc*s, $\frac{2}{3}$ when it is followed by one *acc*, and $\frac{1}{3}$ if it is followed by one *acc* with a delay), and each detection of an obstacle should be followed by a deceleration during two transactions, with a (higher) preference to the first. Clearly, ψ is not realizable with satisfaction value 1, as for some input sequences, namely those with simultaneous or successive occurrences of *safe* and *obs*, it is impossible to respond with the desired patterns of acceleration or declaration. Existing frameworks for synthesis cannot handle this challenge. Indeed, we do not want to add an assumption about *safe* and *obs* occurring far apart. Rather, we want our autonomous car to behave in an optimal way also in problematic environments, and we want, when we evaluate the quality of a car, to take into an account the challenge posed by the environment. This is exactly what high-quality GE-synthesis does: for each input sequence, it requires the synthesized car to obtain the maximal satisfaction value that is possible for that input sequence.

We show that in the Boolean setting, GE-synthesis can be reduced to synthesis of LTL with quantification of atomic propositions [26]. Essentially, GE-synthesis of ψ amounts to synthesis of $(\exists O.\psi) \rightarrow \psi$. We show that by carefully switching between nondeterminisitc and universal automata, we can solve the GE-synthesis problem in doubly-exponential time, thus it is not harder than traditional synthesis. Also, our algorithm is *Safraless*, thus no determinization and parity games are needed [15,17].

A drawback of GE-synthesis is that we do not actually know whether the specification is satisfied. We describe two ways to address this drawback. The first goes beyond providing satisfaction information and enables the designer to partition the specification into a *strong* component, which is guaranteed to be satisfied in all environments, and a *weak* component, which is guaranteed to be satisfied only in hopeful ones. The second way augments GE-realizing systems

by "satisfaction indicators". For example, we show that when a system is lucky to interact with an environment that generates a prefix of an input sequence such that, when combined with a suitable prefix of an output sequence, the specification becomes realizable, then GE-synthesis guarantees that the system indeed responds with a suitable prefix of an output sequence. Moreover, it is easy to add to the system a monitor that detects such prefixes, thus indicating that the specification is going to be satisfied in all environments. Additional monitors we suggest detect prefixes after which the satisfaction becomes valid or unsatisfiable.

We continue to the quantitative setting. We parameterize hope by a satisfaction value $v \in [0,1]$ and say that an input sequence $x \in (2^I)^\omega$ is *v-hopeful* for an LTL[$\mathcal{F}$] formula ψ if an interaction with it can generate a computation that satisfies ψ with value at least v. Formally, there is an output sequence $y \in (2^O)^\omega$ such that $[\![x \otimes y, \psi]\!] \geq v$, where for a computation $w \in (2^{I \cup O})^\omega$, we use $[\![w, \psi]\!]$ to denotes the satisfaction value of ψ in w. As we elaborate below, while the basic idea of GE-synthesis, namely "input sequences with a potential to high quality should realize this potential" is as in the Boolean setting, there are several ways to implement this idea.

We start with a worst-case approach. There, a strategy $f : (2^I)^+ \to 2^O$ GE-realizes an LTL[$\mathcal{F}$] formula ψ if for all input sequences $x \in (2^I)^\omega$, if x is v-hopeful, then $[\![x \otimes f(x), \psi]\!] \geq v$. The requirement can be applied to a threshold value or to all values $v \in [0,1]$. For example, our autonomous car controller has to achieve satisfaction value 1 in roads with no simultaneous or successive occurrences of *safe* and *obs*, and value $\frac{3}{4}$ in roads that violate the latter only with some *obs* followed by *safe*. We then argue that the situation is similar to that of *high-quality assume guarantee synthesis* [3], where richer relations between a quantitative assumption and a quantitative guarantee are of interest. In our case, the assumption is the hopefulness level of the input sequence, namely $[\![x, \exists O.\psi]\!]$, and the guarantee is the satisfaction value of the specification in the generated computation, namely $[\![x \otimes f(x), \psi]\!]$. When synthesizing, for example, a robot controller (e.g., vacuum cleaner) in a building, the doors to rooms are controlled by the environment, whereas the movement of the robot by the system. A measure of the performance of the robot has to take into an account both the number of "hopeful rooms", namely these with an open door – a projection of this number on $[0,1]$ serves as the assumption, and the number of room cleaned – which induces the guarantee. We assume that the desired relation between the assumption and the guarantee is given by a function $\texttt{comb} : [0,1] \times [0,1] \to [0,1]$, which can capture implication, difference, or ratio.

We continue with an analysis of the expected performance of the system. We do so by assuming a stochastic environment, with a known distribution on the input sequences. We introduce and study two measures for high-quality GE-synthesis in a stochastic environment. In the first, termed *expected* GE*-synthesis*, all input sequences are sampled, yet the satisfaction value in each input sequence takes its hopefulness level into account, for example by a `comb` function as in the assume-guarantee setting. In the second, termed *conditional expected*

GE-*synthesis*, only hopeful input sequences are sampled. For both approaches, our synthesis algorithm is based on the high-quality LTL[$\mathcal{F}$] synthesis algorithm of [2], which is based on an analysis of deterministic automata associated with the different satisfaction values of the LTL[$\mathcal{F}$] specification. Here too, the complexity stays doubly exponential. In addition, we extend the synthesized systems with guarantees for satisfaction and monitors indicating satisfaction in various satisfaction levels.

2 Preliminaries

Consider two finite sets I and O of input and output signals, respectively. For two words $x = i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$ and $y = o_0 \cdot o_1 \cdot o_2 \cdots \in (2^I)^\omega$, we define $x \otimes y$ as the word in $(2^{I \cup O})^\omega$ obtained by merging x and y. Thus, $x \otimes y = (i_0 \cup o_0) \cdot (i_1 \cup o_1) \cdot (i_2 \cup o_2) \cdots$. The definition is similar for finite x and y of the same length. For a word $w \in (2^{I \cup O})^\omega$, we use $w_{|I}$ to denote the projection of w on I. In particular, $(x \otimes y)_{|I} = x$.

A *strategy* is a function $f : (2^I)^+ \to 2^O$. Intuitively, f models the interaction of a system that generates in each moment in time a letter in 2^O with an environment that generates letters in 2^I. For an input sequence $x = i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^\omega$, we use $f(x)$ to denote the output sequence $f(i_0) \cdot f(i_0 \cdot i_1) \cdot f(i_0 \cdot i_1 \cdot i_2) \cdots \in (2^O)^\omega$. Then, $x \otimes f(x) \in (2^{I \cup O})^\omega$ is the *computation* of f on x. Note that the environment initiates the interaction, by inputting i_0. Of special interest are *finite-state strategies*, induced by finite state transducers. Formally, an I/O-*transducer* is $\mathcal{T} = \langle I, O, S, s_0, M, \tau \rangle$, where S is a finite set of states, $s_0 \in S$ is an initial state, $M : S \times 2^I \to S$ is a transition function, and $\tau : S \to 2^O$ is a labelling function. For $x = i_0 \cdot i_1 \cdot i_2 \cdots \in (2^I)^*$, let $M^*(x)$ be the state in S that $\mathcal{T}$ reaches after reading x. Thus is, $M^*(\epsilon) = s_0$ and for every $j \geq 0$, we have that $M^*(i_0 \cdot i_1 \cdot i_2 \cdots i_j) = M(M^*(i_0 \cdot i_1 \cdot i_2 \cdots i_{j-1}), i_j)$. Then, $\mathcal{T}$ induces the strategy $f_{\mathcal{T}} : (2^I)^+ \to 2^O$, where for every $x \in (2^I)^+$, we have that $f_{\mathcal{T}}(x) = \tau(M^*(x))$. We use $\mathcal{T}(x)$ and $x \otimes \mathcal{T}(x)$ to denote the output sequence and the computation of $\mathcal{T}$ on x, respectively, and talk about $\mathcal{T}$ realizing a specification, referring to the strategy $f_{\mathcal{T}}$.

We specify on-going behaviors of reactive systems using the *linear temporal logic* LTL [19]. Formulas of LTL are constructed from a set AP of atomic proposition using the usual Boolean operators and temporal operators like G ("always"), F ("eventually"), X ("next time"), and U ("until"). Each LTL formula ψ defines a language $L(\psi) = \{w : w \models \psi\} \subseteq (2^{AP})^\omega$. We also use *automata on infinite words* for specifying and reasoning about on-going behaviors. We use automata with different branching modes (nondeterministic, where some run has to be accepting; universal, where all runs have to be accepting; and deterministic, where there is a single run) and different acceptance conditions (Büchi, co-Büchi, and parity). We use the three letter acronyms NBW, UCW, DPW, and DFW, to refer to nondeterministic Büchi, universal co-Büchi, deterministic parity, and deterministic finite word automata, respectively. Given an LTL formula ψ over AP, one can constructs an NBW $\mathcal{A}_\psi$ with at most $2^{O(|\psi|)}$ states

such that $L(\mathcal{A}_\psi) = L(\psi)$ [27]. Constructing an NBW for $\neg\psi$ and then dualizing it, results in a UCW for $L(\psi)$, also with at most $2^{O(|\psi|)}$ states. Determinization [23] then leads to a DPW for $L(\psi)$ with at at most $2^{2^{O(|\psi|)}}$ states and index $2^{O(|\psi|)}$. For full definitions of LTL, automata, and their relation, see [12].

Consider an LTL formula ψ over $I \cup O$. We say that ψ is *realizable* if there is a finite-state strategy $f : (2^I)^+ \to 2^O$ such that for all $x \in (2^I)^\omega$, we have that $x \otimes f(x) \models \psi$. That is, the computation of f on every input sequence satisfies ψ. We say that a word $x \in (2^I)^\omega$ is *hopeful* for ψ if there is $y \in (2^O)^\omega$ such that $x \otimes y \models \psi$. Then, we say that ψ is *good-enough realizable* (GE-realizable, for short) if there is a finite-state strategy $f : (2^I)^+ \to 2^O$ such that for every $x \in (2^I)^\omega$ that is hopeful for ψ, we have that $x \otimes f(x) \models \psi$. That is, if there is some output sequence whose combination with x satisfies ψ, then the computation of f on x satisfies ψ. The LTL GE-synthesis problem is then to decide whether a given LTL formula is GE-realizable, and if so, to return a transducer that GE-realizes it. Clearly, every realizable specification is GE-realizable – by the same transducer. We say that ψ is *universally satisfiable* if all input sequences are hopeful for ψ. It is easy to see that for universally satisfiable specifications, realizability and GE-realizability coincide. On the other hand, as demonstrated in Sect. 1, there are specifications that are not realizable and are GE-realizable.

Example 1. Let $I = \{p\}$ and $O = \{q\}$. Consider the specification $\psi = \mathsf{GF}((\mathsf{X}p) \wedge q) \wedge \mathsf{GF}((\mathsf{X}\neg p) \wedge \neg q)$. Clearly, ψ is not realizable, as an input sequence $x \in (2^I)^\omega$ is hopeful for ψ iff $x \models \mathsf{GF}p \wedge \mathsf{GF}\neg p$. Since the system has to assign a value to q before it knowns the value of $\mathsf{X}p$, it seems that ψ is also not GE-realizable. As we show below, however, the specification ψ is GE-realizable. Intuitively, it follows from the fact that hopeful input sequences consists of alternating p-blocks and $(\neg p)$-blocks. Then, by outputting $\neg q$ in p-blocks and outputting q in $(\neg p)$-blocks, the system guarantees that each last position in a $(\neg p)$-block satisfies $q \wedge \mathsf{X}p$ and each last position in a p-block satisfies $(\neg q) \wedge \mathsf{X}p$. Formally, ψ is GE-realized by the transducer $\mathcal{T} = \langle \{p\}, \{q\}, \{s_0, s_1\}, s_0, M, \tau \rangle$, where $M(s_0, \emptyset) = M(s_1, \emptyset) = s_0$, $M(s_0, \{p\}) = M(s_1, \{p\}) = s_1$, $\tau(s_0) = \{q\}$, and $\tau(s_1) = \emptyset$. □

3 LTL Good-Enough Synthesis

Recall that a strategy $f : (2^I)^+ \to 2^O$ GE-realizes an LTL formula ψ if its computations on all hopeful input sequences satisfy ψ. Thus, for every input sequence $x \in (2^I)^\omega$, either $x \otimes y \not\models \psi$ for all $y \in (2^O)^\omega$, or $x \otimes f(x) \models \psi$. The above suggests that algorithms for solving LTL GE-synthesis involve existential and universal quantification over the behavior of output signals. The logic EQLTL extends LTL by allowing existential quantification over atomic propositions [26]. We refer here to the case the atomic propositions are the signals in $I \cup O$, and the signals in O are existentially quantified. Then, an EQLTL formula is of the form $\exists O.\psi$, and a computation $w \in (2^{I \cup O})^\omega$ satisfies $\exists O.\psi$ iff there is $y \in (2^O)^\omega$ such that $w_{|I} \otimes y \models \psi$. Dually, AQLTL extends LTL by allowing universal quantification over atomic propositions. We consider here formulas of the form

$\forall O.\psi$, which are equivalent to $\neg\exists O.\neg\psi$. Indeed, a computation $w \in (2^{I\cup O})^\omega$ satisfies $\forall O.\psi$ iff for all $y \in (2^O)^\omega$, we have that $w_{|I} \otimes y \models \psi$. Note that in both the existential and universal cases, the O-component of w is ignored. Accordingly, we sometimes interpret EQLTL and AQLTL formulas with respect to input sequences $x \in (2^I)^\omega$. Also note that both EQLTL and AQLTL increase the expressive power of LTL. For example, the EQLTL formula $\exists q.q \wedge \mathsf{X}\neg q \wedge \mathsf{G}(q \leftrightarrow \mathsf{XX}q) \wedge \mathsf{G}(q \rightarrow p)$ states that p holds in all even positions of the computation, which cannot be specified in LTL [29].

Theorem 1. *The LTL* GE*-synthesis problem is 2EXPTIME-complete.*

Proof. We start with the upper bound. Given an LTL formula ψ over $I \cup O$, we describe an algorithm that returns a transducer $\mathcal{T}$ that GE-realizes ψ, or declares that no such transducer exists.

It is not hard to see that $\mathcal{T}$ GE-realizes ψ iff $\mathcal{T}$ realizes $\varphi = \psi \vee \forall O.\neg\psi$. Indeed, an input sequence $x \in (2^I)^\omega$ is hopeful for ψ iff $x \models \exists O.\psi$, and so the specification φ requires all hopeful input sequences to satisfy ψ. A naive construction of an NBW for φ involves a universal projection of the signals in O in an automaton for $\neg\psi$, and results in an NBW that is doubly exponential. In order to circumvent the extra exponent, we construct an NBW $\mathcal{A}_{\neg\varphi}$ for $\neg\varphi$, and then dualize it to get a UCW for φ, as follows.

Let $\mathcal{A}_{\neg\psi}$ be an NBW for $L(\neg\psi)$ and $\mathcal{A}_{\exists O.\psi}$ be an NBW for $L(\exists O.\psi)$. Thus, $\mathcal{A}_{\exists O.\psi}$ is obtained from an NBW $\mathcal{A}_\psi$ for $L(\psi)$ by existentially projecting its transitions on 2^I. In more details, if $\mathcal{A}_\psi = \langle 2^{I\cup O}, Q, Q_0, \delta, \alpha\rangle$, then $\mathcal{A}_{\exists O.\psi} = \langle 2^{I\cup O}, Q, Q_0, \delta', \alpha\rangle$, where for all $q \in Q$ and $i \cup o \in 2^{I\cup O}$, we have $\delta'(q,\sigma) = \bigcup_{o\in 2^O}\{\delta(q,(\sigma \cap I) \cup o)\}$.

Let $\mathcal{A}_{\neg\varphi}$ be an NBW for the intersection of $\mathcal{A}_{\neg\psi}$ and $\mathcal{A}_{\exists O.\psi}$. We can define $\mathcal{A}_{\neg\varphi}$ as the product of $\mathcal{A}_{\neg\psi}$ and $\mathcal{A}_{\exists O.\psi}$, possibly using the generalized Büchi acceptance condition (see Remark 1), thus its size is exponential in ψ. The language of $\mathcal{A}_{\neg\varphi}$ is then $\{w \in (2^{I\cup O})^\omega : w \not\models \psi \text{ and } w \models \exists O.\psi\}$. We then solve usual synthesis for the complementing UCW. Its language is $\{w \in (2^{I\cup O})^\omega : w \models \psi \text{ or } w \models \forall O.\neg\psi\}$, as required. By [17], the synthesis problem for UCW can be solved in EXPTIME, and we are done.

The lower bound follows from the 2EXPTIME-hardness of LTL realizability [22]. The hardness proof there constructs, given a 2EXPTIME Turing machine M, an LTL formula ψ that is realizable iff M accepts the empty tape. Since all input sequences are hopeful for ψ, realizability and GE-realizability coincide, and we are done. □

Note that working with a UCW not only handles the universal quantification for free but also has the advantage of a Safraless synthesis algorithm – no determinization and parity games are needed [15,17]. Also note that the algorithm we suggest in the proof of Theorem 1 can be generalized to handle specifications that are arbitrary positive Boolean combinations of EQLTL formulas.

Remark 1 [**Products and optimizations**]. Throughout the paper, we construct products of automata whose state space is $2^{cl(\psi)}$, and states correspond

to maximal consistent subsets of $cl(\psi)$, possibly in the scope of an existential quantifier of O. Accordingly, the product can be minimized to include only consistent pairs. Also, since traditional-synthesis algorithms, in particular the Safra-less algorithms we use, can handle automata with *generalized* Büchi and co-Büchi acceptance condition, we need only one copy of the product. □

Remark 2 [**Determinancy of the** GE**-synthesis game**]. Determinancy of games implies that in traditional synthesis, a specification ψ is not I/O-realizable iff $\neg\psi$ is O/I-realizable This is useful, for example when we want to synthesize a transducer of a bounded size and proceed simultaneously, aiming to synthesize either a system transducer that realizes ψ or an environment transducer that realizes $\neg\psi$ [17]. For GE-synthesis, simple dualization does not hold, but we do have determinancy in the sense that $(\exists O.\psi) \rightarrow \psi$ is not I/O-realizable iff $(\exists O.\psi) \wedge \neg\psi$ is O/I-realizable. Accordingly, ψ is not GE-realizable iff the environment has a strategy that generates, for each output sequence $y \in (2^O)^\omega$, a helpful input sequence $x \in (2^I)^\omega$ such that $x \otimes y \models \neg\psi$. In the full version, we formalize and study this duality further. □

4 Guarantees in Good-Enough Synthesis

A drawback of GE-synthesis is that we do not actually know whether the specification is satisfied. In this section we describe two ways to address this drawback. The first way goes beyond providing satisfaction information and enables the designer to partition the specification into to a *strong* component, which should be satisfied in all environments, and a *weak* component, which should be satisfied only in hopeful ones. The second way augments GE-realizing transducers by flags, raised to indicate the status of the satisfaction.

4.1 GE-Synthesis with a Guarantee

Recall that GE-realizability is suitable especially in settings where we design a system that has to do its best in all environments. GE-synthesis with a guarantee is suitable in settings where we want to make sure that some components of the specification are satisfied in all environment. Accordingly, a specification is an LTL formula $\psi = \psi_{strong} \wedge \psi_{weak}$. When we GE-*synthesize* ψ_{weak} *with guarantee* ψ_{strong}, we seek a transducer $\mathcal{T}$ that realizes ψ_{strong} and GE-realizes ψ_{weak}. Thus, for all input sequences $x \in (2^I)^\omega$, we have that $x \otimes \mathcal{T}(x) \models \psi_{strong}$, and if x is hopeful for ψ_{weak}, then $x \otimes \mathcal{T}(x) \models \psi_{strong}$.

Theorem 2. *The LTL* GE*-synthesis with guarantee problem is 2EXPTIME-complete.*

Proof. Consider an LTL formula $\psi = \psi_{strong} \wedge \psi_{weak}$ over $I \cup O$. It is not hard to see that a transducer $\mathcal{T}$ GE-realizes ψ_{weak} with guarantee ψ_{strong} iff $\mathcal{T}$ realizes $\varphi = \psi_{strong} \wedge ((\exists O.\psi_{weak}) \rightarrow \psi_{weak})$. We can then construct a UCW $\mathcal{A}_\varphi$ for $L(\varphi)$ by dualizing an NBW for its negation $\neg\psi_{strong} \vee ((\exists O.\psi_{weak}) \wedge \neg\psi_{weak})$, which

can be constructed using techniques similar to those in the proof of Theorem 1. We then proceed with standard synthesis for $\mathcal{A}_\varphi$. Note that the approach is Safraless. Taking an empty (that is, `True`) guarantee, a lower bound follows from the 2EXPTIME-hardness of LTL GE-synthesis. □

4.2 Flags by a GE-Realizing Transducer

For a language $L \subseteq (2^{I \cup O})^\omega$ and a finite word $w \in (2^{I \cup O})^*$, let $L^w = \{w' \in (2^{I \cup O})^\omega : w \cdot w' \in L\}$. That is, L^w is the language of suffixes of words in L that have w as a prefix. We say that a word $w \in (2^{I \cup O})^*$ is *green for* L if L^w is realizable. Then, a word $x \in (2^I)^*$ is *green for* L if there is $y \in (2^O)^*$ such that $x \otimes y$ is green for L. When a system is lucky to interact with an environment that generates a green input sequence, we want the system to react in a way that generates a green prefix, and then realizes the specification. Formally, we say that a strategy $f : (2^I)^+ \to 2^O$ *green realizes* L if for every $x \in (2^I)^+$, if x is green for L, then $x \otimes f(x)$ is green for L.[1,2] We say that a word $w \in (2^{I \cup O})^*$ is *light green for* L if L^w is universally satisfiable, thus all input sequences are hopeful for L^w. A word $x \in (2^I)^*$ is *light green for* L if there is $y \in (2^O)^*$ such that $x \otimes y$ is light green for L. It is not hard to see that for GE-realizable languages, green and light green coincide. Indeed, if L is universally satisfiable and GE-realizable, then L is realizable.

Theorem 3. GE*-realizability is strictly stronger than green realizability.*

Proof. We first prove that every strategy $f : (2^I)^+ \to 2^O$ that GE-realizes a specification ψ also green realizes ψ. Consider $x \in (2^I)^+$ that is green for ψ. By definition, there is $y \in (2^O)^+$ such that $L^{x \otimes y}$ is realizable. Then, for every $x' \in (2^I)^\omega$, there is $y' \in (2^O)^\omega$ such that $x' \otimes y'$ in $L^{x \otimes y}$. Hence, for every $x' \in (2^I)^\omega$, we have that $x \cdot x'$ is hopeful. Therefore, as f GE-realizes ψ, we have that $(x \cdot x') \otimes f(x \cdot x') \models \psi$. Thus, $x \otimes f(x)$ is green, and so f green realizes ψ.

We continue and describe a specification that is green realizable and not GE-realizable. Let $I = \{p\}$ and $O = \{q\}$. Consider the specification $\psi = \mathsf{G}((\mathsf{X}p) \leftrightarrow q)$. Clearly, ψ is not realizable, as the system has to commit a value for q before a value for Xp is known. Likewise, no word $w \in (2^{I \cup O})^*$ is green for ψ, and so no finite input sequence $x \in (2^I)^*$ is green for ψ. Hence, every strategy (vacuously) green realizes ψ. On the other hand, for every input sequences $x \in (2^I)^\omega$ there is an output sequence $y \in (2^O)^\omega$ such that $x \otimes y \models \psi$. Thus, all input sequences are hopeful for ψ. Thus, synthesis and GE-synthesis coincide for ψ, which is not GE-realizable. □

Theorem 3 brings with it two good news. The first is that a GE-realizing transducer has the desired property of being also green realizing. The second has

[1] Note that while the definition of green realization does not refer to ϵ directly, we have that ϵ is green iff L is realizable, in which case all $x \in (2^I)^*$ are green.

[2] While synthesis corresponds to finding a winning strategy for the system, green synthesis can be viewed as a subgame-perfect best-response strategy, where the system does its best in every subgame, even if it loses the overall game.

to do with our goal of providing the user with information about the satisfaction status, in particular raising a green flag whenever a green prefix is detected. By Theorem 3, such a flag indicates that the computation generated by our GE-realizing transducer satisfies the specification. A naive way to detect green prefixes for a specification ψ is to solve the synthesis problem for ψ by solving a game on top of a DPW $\mathcal{D}_\psi$ for ψ. The winning positions in the game are states in $\mathcal{D}_\psi$. By defining them as accepting states, we can obtain from $\mathcal{D}_\psi$ a DFW for green prefixes. Then, we run this DFW in parallel with the GE-realizing transducer, and raise the green flag whenever a green prefix is detected. This, however, requires a generation of $\mathcal{D}_\psi$ and a solution of parity games. Below we describe a much simpler way, which makes use of the fact that our transducer GE-realizes the specification.

Recall that if L is universally satisfiable and GE-realizable, then L is realizable. Accordingly, given a transducer $\mathcal{T}$ that GE-realizes ψ, we can augment it with green flags by running in parallel a DFW that detects light-green prefixes. As we argue below, constructing such a DFW only requires an application of the subset construction on top of an NBW for the existential projection of ψ on 2^I.

Lemma 1. *Given an LTL formula ψ over $I \cup O$, we can construct a DFA $\mathcal{S}$ of size $2^{2^{O(|\psi|)}}$ such that $L(\mathcal{S}) = \{x \in (2^I)^* : x \text{ is light green for } L(\psi)\}$.*

Proof. Let $\mathcal{A}_\psi = \langle 2^{I \cup O}, Q, \delta, Q_0, \alpha \rangle$ be an NBW for $L(\psi)$, and let $\mathcal{B}_\psi = \langle 2^I, Q, \delta', Q_0, \alpha \rangle$ be its existential projection on 2^I. Thus, for every $q \in Q$ and $i \in 2^I$, we have $\delta'(q, i) = \bigcup_{o \in 2^O} \delta(q, i \cup o)$. We define the DFW $\mathcal{S} = \langle 2^I, 2^Q, M, \{Q_0\}, F \rangle$, where M follows the subset construction of $\mathcal{B}_\psi$: for every $S \in 2^Q$ and $i \in 2^I$, we have $M(S, i) = \bigcup_{s \in S} \delta'(s, i)$. Then, $F = \{S \in 2^Q : L(\mathcal{B}^S_\psi) = (2^I)^\omega\}$. Observe that $\mathcal{S}$ rejects $x \in (2^I)^*$ iff there is $x' \in (2^I)^\omega$ such that for all $y \in (2^O)^*$ and $y' \in (2^O)^\omega$, no state in $\delta(Q_0, x \otimes y)$ accepts $x' \otimes y'$. Thus, $\mathcal{S}$ rejects x iff x is not light green, and accepts it otherwise. Note that the definition of F involves universality checking, possibly via complementation, yet no determinization is required, and the size of $\mathcal{S}$ is $2^{2^{O(|\psi|)}}$. □

Note that once we reach an accepting state in $\mathcal{S}$, we can make it an accepting loop. Indeed, once a green prefix is detected, then all prefixes that extend it are green. Accordingly, once the green flag is raised, it stays up. Also note that if an input sequence is not hopeful for ψ, then none of its prefixes is light green for ψ. The converse, however, is not true: an input sequence may be hopeful and still have no light green prefixes. For example, taking $I = \{p\}$, the input sequence $\{p\}^\omega$ is hopeful for $\mathsf{G}p$, yet none of its prefixes is green light, as it can be extended to an input sequence with $\neg p$.

Green flags provide information about satisfaction. Two additional flags of interest are related to safety and co-safety properties:

- A word $w \in (2^{I \cup O})^*$ is *red for* L if $L^w = \emptyset$. A word $x \in (2^I)^*$ is *red for* L if for all $y \in (2^O)^*$, we have that $x \otimes y$ is red for L. Thus, when the environment generates x, then no matter how the system responds, L is not satisfied.

– a word $w \in (2^{I \cup O})^*$ is *blue for* L when $L^w = (2^{I \cup O})^\omega$, and then define a word $x \in (2^I)^*$ as *blue for* L if there is $y \in (2^O)^*$ such that $x \otimes y$ is blue for L. Thus, when the environment generates x, the system can respond in a way that guarantees satisfaction no matter how the interaction continues.

A monitor that detects red and blue prefixes for L can be added to a transducer that GE-realizes L. As has been the case with the monitor for green prefixes, its construction is based on applying the subset construction on an NBW for L [16]. Also, once a red or blue flag is raised, it stays up. In a way analogous to green realizability, we seek a transducer that GE-realizes the specification and generates a red prefix only if all interactions generate a red prefix, and generates a blue prefix whenever this is possible. In the full version, we show that while GE-realization implies *red realization*, it may conflict with *blue realization*.

5 High-Quality Good-Enough Synthesis

GE-synthesis is of special interest when the satisfaction value of the specification is multi-valued, and we want to synthesize high-quality systems. We start by defining the multi-valued logic LTL[$\mathcal{F}$], which is our multi-valued specification formalism. We then study LTL[$\mathcal{F}$] GE-synthesis, first in a worst-case approach, where the satisfaction value of a transducer is the satisfaction value of its computation with the lowest satisfaction value, and then in a stochastic approach, where it is the expected satisfaction value, given a distribution of the inputs.

5.1 The Logic LTL[$\mathcal{F}$]

Let AP be a set of Boolean atomic propositions and let $\mathcal{F} \subseteq \{f : [0,1]^k \to [0,1] : k \in \mathbb{N}\}$ be a set of *quality operators*. An LTL[$\mathcal{F}$] formula is one of the following:

– $\texttt{True}$, $\texttt{False}$, or p, for $p \in AP$.
– $f(\psi_1, ..., \psi_k)$, $\mathsf{X}\psi_1$, or $\psi_1 \mathsf{U} \psi_2$, for LTL[$\mathcal{F}$] formulas $\psi_1, \ldots, \psi_k$ and a function $f \in \mathcal{F}$.

The semantics of LTL[$\mathcal{F}$] formulas is defined with respect to infinite computations over AP. For a computation $w = w_0, w_1, \ldots \in (2^{AP})^\omega$ and position $j \geq 0$, we use w^j to denote the suffix $w_j, w_{j+1}, \ldots$. The semantics maps a computation w and an LTL[$\mathcal{F}$] formula ψ to the *satisfaction value* of ψ in w, denoted $[\![w, \psi]\!]$. The satisfaction value is in $[0,1]$ and is defined inductively as follows.

– $[\![w, \texttt{True}]\!] = 1$ and $[\![w, \texttt{False}]\!] = 0$.
– For $p \in AP$, we have that $[\![w, p]\!] = 1$ if $p \in w_0$, and $[\![w, p]\!] = 0$ if $p \notin w_0$.
– $[\![w, f(\psi_1, ..., \psi_k)]\!] = f([\![w, \psi_1]\!], ..., [\![w, \psi_k]\!])$.
– $[\![w, \mathsf{X}\psi_1]\!] = [\![w^1, \psi_1]\!]$.
– $[\![w, \psi_1 \mathsf{U} \psi_2]\!] = \max_{i \geq 0}\{\min\{[\![w^i, \psi_2]\!], \min_{0 \leq j < i} [\![w^j, \psi_1]\!]\}\}$.

The logic LTL can be viewed as LTL[$\mathcal{F}$] for $\mathcal{F}$ that models the usual Boolean operators. In particular, the only possible satisfaction values are 0 and 1. We abbreviate common functions as described below. Let $x, y, \lambda \in [0,1]$. Then,

- $\neg x = 1 - x$
- $x \vee y = \max\{x, y\}$
- $x \wedge y = \min\{x, y\}$
- $x \rightarrow y = \max\{1 - x, y\}$
- $\triangledown_\lambda x = \lambda \cdot x$
- $x \oplus_\lambda y = \lambda \cdot x + (1 - \lambda) \cdot y$

The realizability problem for LTL[$\mathcal{F}$] is an optimization problem: For an LTL[$\mathcal{F}$] specification ψ and a transducer $\mathcal{T}$, we define the satisfaction value of ψ in $\mathcal{T}$, denoted $[\![\mathcal{T}, \psi]\!]$, by $\min\{[\![x \otimes \mathcal{T}(x), \psi]\!] : x \in (2^I)^\omega\}$, namely the satisfaction value of ψ in the worst-case. Then, the synthesis problem is to find, given ψ, a transducer that maximizes its satisfaction value. Moving to a decision problem, given ψ and a threshold value $v \in [0, 1]$, we say that ψ is *v-realizable* if there exists a transducer $\mathcal{T}$ such that $[\![\mathcal{T}, \psi]\!] \geq v$, and the synthesis problem is to find, given ψ and v, a transducer $\mathcal{T}$ that v-realizes ψ.

For an LTL[$\mathcal{F}$] formula ψ, let $V(\psi)$ be the set of possible satisfaction values of ψ in arbitrary computations. Thus, $V(\psi) = \{[\![w, \psi]\!] : w \in (2^{AP})^\omega\}$.

Theorem 4 [1]. *Consider an* LTL[$\mathcal{F}$] *formula* ψ.

- $|V(\psi)| \leq 2^{|\psi|}$.
- *For every predicate* $P \subseteq [0, 1]$, *there exists an NBW* $\mathcal{A}^P_\psi$ *such that* $L(\mathcal{A}^P_\psi) = \{w : [\![w, \psi]\!] \in P\}$. *Furthermore,* $\mathcal{A}^P_\psi$ *has at most* $2^{O(|\psi|^2)}$ *states* [1].

As with LTL, we define the existential and universal extensions EQLTL[$\mathcal{F}$] and AQLTL[$\mathcal{F}$] of LTL[$\mathcal{F}$]. Here too, we consider the case $AP = I \cup O$, with the signals in O being quantified. Then, $[\![w, \exists O.\psi]\!] = \max_{y \in (2^O)^\omega}\{[\![w_{|I} \otimes y, \psi]\!]\}$ and $[\![w, \forall O.\psi]\!] = \min_{y \in (2^O)^\omega}\{[\![w_{|I} \otimes y, \psi]\!]\}$.

Remark 3 **[On the semantics of** EQLTL[$\mathcal{F}$]]. It is tempting to interpret an expression like $[\![w, \exists O.\psi]\!] \leq v$ as "there exists an output sequence y such that $[\![w_I \otimes y, \psi]\!] \leq v$". By the semantics of $\exists O.\psi$, however, $[\![w, \exists O.\psi]\!] \leq v$ actually means that $\max_{y \in (2^O)^\omega} [\![w_I \otimes y, \psi]\!] \leq v$. Thus, the correct interpretation is "for all output sequences y, we have that $[\![w_I \otimes y, \psi]\!] \leq v$". □

5.2 LTL[$\mathcal{F}$] GE-Synthesis

For a value $v \in [0, 1]$, we say that x is *v-hopeful for* ψ if there is $y \in (2^O)^\omega$ such that $[\![x \otimes y, \psi]\!] \geq v$. We study two variants of LTL[$\mathcal{F}$] GE-synthesis:

- In LTL[$\mathcal{F}$] GE-*synthesis with a threshold*, the input is an LTL[$\mathcal{F}$] formula ψ and a value $v \in [0, 1]$, and the goal is to generate a transducer whose computation on every input sequence that is v-hopeful has satisfaction value at least v. Formally, a function $f : (2^I)^+ \rightarrow 2^O$ GE-realizes ψ with threshold v if for every $x \in (2^I)^\omega$, if x is v-hopeful, then $[\![x \otimes f(x), \psi]\!] \geq v$.
- In LTL[$\mathcal{F}$] GE-*synthesis*, the input is an LTL[$\mathcal{F}$] formula ψ, and the goal is to generate a transducer whose computation on every input sequence has the highest possible satisfaction value for this input sequence. Formally, a function $f : (2^I)^+ \rightarrow 2^O$ GE-realizes ψ if for every $x \in (2^I)^\omega$ and value $v \in [0, 1]$, if x is v-hopeful, then $[\![x \otimes f(x), \psi]\!] \geq v$.

In the Boolean case, the two variants coincide, taking $v = 1$. Indeed, then, for every $x \in (2^I)^\omega$, if x is hopeful, then $x \otimes f(x)$ has to satisfy ψ. We note that GE-realization with a threshold is not monotone, in the sense that decreasing the threshold need not lead to GE-realization. Indeed, the lower is the threshold v, the more input sequences are v-helpful (see Example 2). Accordingly, we do not search for a maximal threshold, and rather may ask about a desired threshold or about GE-synthesis without a threshold.

Solving the GE-synthesis problem, a naive combination of the automata construction of Theorem 4 with the projection technique of Theorem 1, corresponds to an erroneous semantics of EQLTL[$\mathcal{F}$], as noted in Remark 3. Before describing our construction, it is helpful to state the correct (perhaps less intuitive) interpretation of existential and universal quantification in the quantitative setting:

Lemma 2. *For every* LTL[$\mathcal{F}$] *formula* ψ *and an input sequence* $x \in (2^I)^\omega$*, we have that* $[\![x, \exists O.\psi]\!] = 1 - [\![x, \forall O.\neg\psi]\!]$*. Accordingly, for every value* $v \in [0,1]$*, we have that* $[\![x, \exists O.\psi]\!] < v$ *iff* $[\![x, \forall O.\neg\psi]\!] > 1 - v$*.*

Proof. By definition, $[\![x, \exists O.\psi]\!] = \max_{y \in (2^O)^\omega} [\![x \otimes y, \psi]\!] = 1 - \min_{y \in (2^O)^\omega} 1 - [\![x \otimes y, \psi]\!] = 1 - \min_{y \in (2^O)^\omega} [\![x \otimes y, \neg\psi]\!] = 1 - [\![x, \forall O.\neg\psi]\!]$. Then, $[\![x, \exists O.\psi]\!] < v$ iff $1 - [\![x, \exists O.\psi]\!] > 1 - v$ iff $[\![x, \forall O.\neg\psi]\!] > 1 - v$. □

Consider an LTL[$\mathcal{F}$] formula ψ, a value $v \in [0,1]$, and an input sequence $x \in (2^I)^\omega$. Recall that x is v-hopeful for ψ if there is $y \in (2^O)^\omega$ such that $[\![x \otimes y, \psi]\!] \geq v$. Equivalently, $[\![x, \exists O.\psi]\!] \geq v$. Indeed, $[\![x, \exists O.\psi]\!] = \max_{y \in (2^O)^\omega} [\![x \otimes y, \psi]\!]$, which is greater or equal to v iff there is $y \in (2^O)^\omega$ such that $[\![x \otimes y, \psi]\!] \geq v$. Hence, x is not v-hopeful for ψ if $[\![x, \exists O.\psi]\!] < v$. Equivalently, by Lemma 2, $[\![x, \forall O.\neg\psi]\!] > 1 - v$. Accordingly, for a strategy $f : (2^I)^+ \to 2^O$, an input sequence $x \in (2^I)^\omega$, and a value $v \in [0,1]$, we say that f is *v-good for x with respect to ψ*, if $[\![x \otimes f(x), \psi]\!] \geq v$ or $[\![x, \forall O.\neg\psi]\!] > 1 - v$.

Example 2. Let $I = \{p\}$ and $O = \{q\}$. Consider the LTL[$\mathcal{F}$] formula $\psi = (\nabla_{\frac{1}{4}} p \vee \nabla_{\frac{1}{2}} q)$. Checking for which values v a strategy f is v-good for x with respect to ψ, we examine whether $[\![x \otimes f(x), \nabla_{\frac{1}{4}} p \vee \nabla_{\frac{1}{2}} q]\!] \geq v$ or $[\![x, \forall q.\neg(\nabla_{\frac{1}{4}} p \vee \nabla_{\frac{1}{2}} q)]\!] > 1 - v$. Since ψ refers only to the first position in the computation, it is enough to examine x_0 and $f(x_0)$. For example, if $x_0 = \emptyset$ and $f(x_0) = \emptyset$, then $[\![x \otimes f(x), \nabla_{\frac{1}{4}} p \vee \nabla_{\frac{1}{2}} q]\!] = 0$, $[\![x, \exists q.\nabla_{\frac{1}{4}} p \vee \nabla_{\frac{1}{2}} q]\!] = \max\{0, \frac{1}{2}\} = \frac{1}{2}$, and $[\![x, \forall q.\neg(\nabla_{\frac{1}{4}} p \vee \nabla_{\frac{1}{2}} q)]\!] = \min\{1, 1 - \frac{1}{2}\} = \frac{1}{2}$. Hence, f is v-good for x with respect to ψ if $v = 0$ or $v > \frac{1}{2}$, thus $v \in \{0\} \cup (\frac{1}{2}, 1]$. Similarly, we have the following.

- If $x_0 = \emptyset$ and $f(x_0) = \{q\}$ then f is v-good for x when $v \in [0,1]$.
- If $x_0 = \{p\}$ and $f(x_0) = \emptyset$ then f is v-good for x when $v \in [0, \frac{1}{4}] \cup (\frac{1}{2}, 1]$.
- If $x_0 = \{p\}$ and$f(x_0) = \{q\}$ then f is v-good for x when $v \in [0,1]$.

Theorem 5. *The* LTL[$\mathcal{F}$] GE*-synthesis with threshold problem is 2EXPTIME-complete.*

Proof. We show we can adjust the upper bound described in the proof of Theorem 1 to the multi-valued setting. Given an LTL[$\mathcal{F}$] formula ψ over $I \cup O$ and a threshold $v \in [0,1]$, we describe an algorithm that returns a transducer $\mathcal{T}$ that GE-realizes ψ with threshold v, or declares that no such transducer exists.

By definition, we have that $\mathcal{T}$ GE-realizes ψ with threshold v if for every input sequence x, we have that $f_{\mathcal{T}}$ is v-good for x with respect to ψ. Thus, $[\![x \otimes f_{\mathcal{T}}(x), \psi]\!] \geq v$ or $[\![x, \forall O.\neg\psi]\!] > 1 - v$. We construct a UCW whose language is $\{w \in (2^{I\cup O})^\omega : [\![w, \psi]\!] \geq v \text{ or } [\![w, \forall O.\neg\psi]\!] > 1 - v\}$.

Let $\mathcal{A}^{<v}_{\psi}$ be an NBW for $\{w : [\![w, \psi]\!] < v\}$ and $\mathcal{A}^{\geq v}_{\exists O.\psi}$ be an NBW for $\{w : [\![w, \exists O.\psi]\!] \geq v\}$. Thus, $\mathcal{A}^{\geq v}_{\exists O.\psi}$ is obtained from an NBW $\mathcal{A}^{\geq v}_{\psi}$ for $\{w : [\![w, \psi]\!] \geq v\}$ by existentially projecting its transitions on 2^I. By Theorem 4, both $\mathcal{A}^{<v}_{\psi}$ and $\mathcal{A}^{\geq v}_{\exists O.\psi}$ are of size exponential in ψ.

Let $\mathcal{B}^{v}_{\psi}$ be an NBW for the intersection of $\mathcal{A}^{<v}_{\psi}$ and $\mathcal{A}^{\geq v}_{\exists O.\psi}$. The language of B^{v}_{ψ} is then $\{w \in (2^{I\cup O})^\omega : [\![w, \psi]\!] < v \text{ and } [\![w, \exists O.\psi]\!] \geq v\}$. We then solve usual synthesis for the complementing UCW, whose language is $\{w \in (2^{I\cup O})^\omega : [\![w, \psi]\!] \geq v \text{ or } [\![w, \forall O.\neg\psi]\!] > 1 - v\}$, as required. By [17], the synthesis problem for UCW can be solved in EXPTIME.

The lower bound follows from the 2EXPTIME-hardness of LTL GE-realizability. □

Theorem 6. *The* LTL[$\mathcal{F}$] GE*-synthesis problem is 2EXPTIME-complete.*

Proof. We start with the upper bound. Given an LTL[$\mathcal{F}$] specification ψ over $I \cup O$, we describe an algorithm that returns a transducer $\mathcal{T}$ that GE-realizes ψ or declares that no such transducer exists.

As discussed above, a transducer $\mathcal{T}$ GE-realizes ψ iff for every input sequence $x \in (2^I)^\omega$ and value $v \in [0,1]$, we have that $f_{\mathcal{T}}$ is v-good for x with respect to ψ. Accordingly, we construct a UCW whose language is $\bigcap_{v \in V(\psi)} \{w \in (2^{I\cup O})^\omega : [\![w, \psi]\!] \geq v \text{ or } [\![w, \forall O.\neg\psi]\!] > 1 - v\}$.

For $v \in V(\psi)$, let $\mathcal{B}^{v}_{\psi}$ be an NBW for $\{w : [\![w, \neg\psi]\!] \geq v \text{ and } [\![w, \exists O.\psi]\!] \geq v\}$, as constructed in the proof of Theorem 5, and let $\mathcal{B}$ be the union of $\mathcal{B}^{v}_{\psi}$ for all $v \in V(\psi)$. By Theorem 4, the size of $V(\psi)$ is exponential in ψ, and thus so is the size of $\mathcal{B}$. We then solve usual synthesis for the complementing UCW, whose language is as required. By [17], the synthesis problem for UCW can be solved in EXPTIME. The lower bound follows from the 2EXPTIME-hardness of LTL GE-realizability. □

Remark 4 **[Tuning hope down]**. The quantitative setting allows the designer to tune down "satisfaction by hoplessness": rather than synthesizing $\psi \vee \forall O.\neg\psi$, we can have a factor λ and synthesize $\psi \vee \nabla_\lambda \forall O.\neg\psi$. In Sect. 5.3 below we study additional ways to refer to hopefulness levels.

5.3 LTL[$\mathcal{F}$] Assume-Guarantee GE-Synthesis

In Sect. 5.2, we seek a transducer $\mathcal{T}$ such that for a given or for all values $v \in [0,1]$ and input sequences $x \in (2^I)^\omega$, if $[\![x, \exists O.\psi]\!] \geq v$ then $[\![x \otimes \mathcal{T}(x), \psi]\!] \geq v$. In this

section we measure the quality of a transducer $\mathcal{T}$ by analyzing richer relations between $[\![x, \exists O.\psi]\!]$ and $[\![x \otimes \mathcal{T}(x), \psi]\!]$. The setting has the flavor of quantitative assume-guarantee synthesis [3]. There, the specification consists of a multi-valued assumption A, which in our case is $\exists O.\psi$, and a multi-valued guarantee G, which is our case is ψ.

There are different ways to analyze the relation between $[\![x, \exists O.\psi]\!]$ and $[\![x \otimes \mathcal{T}(x), \psi]\!]$. To this end, we assume that we are given a function $\texttt{comb} : [0,1] \times [0,1] \to [0,1]$ that given the satisfaction values of $\exists O.\psi$ and of ψ, outputs a combined satisfaction value. We assume that $\texttt{comb}$ is decreasing in the first component and increasing in the second component. This corresponds to the intuition that a lower satisfaction value of $\exists O.\psi$ and a higher satisfaction value of ψ both yield a higher overall score. Also, since $[\![x, \exists O.\psi]\!] \geq [\![x \otimes \mathcal{T}(x), \psi]\!]$ for all $x \in (2^I)^\omega$, we assume that the first component is greater than or equal to the second. Finally, we require $\texttt{comb}$ to be efficiently computed. Some natural $\texttt{comb}$ functions include:

- The quantitative implication function: $\texttt{comb}(A, G) = \max\{1 - A, G\}$. This captures the quantitative notion of the implication $(\exists O.\psi) \to \psi$.
- The (negated) difference function: $\texttt{comb}(A, G) = 1 - (A - G)$. This captures how far the satisfaction value for the given computation is from the best satisfaction value. Since $A \geq G$, the range of the function is indeed $[0, 1]$.
- The ratio function, given by some normalization to $[0, 1]$ of the function $\texttt{comb}(A, G) = \frac{G}{A}$, which captures the "relative success" with respect to the best possible satisfaction value.

The choice of an appropriate $\texttt{comb}$ function depends on the setting. Implication is in order when harsh environments may outweigh the actual performance of the system. For example, if our specification measures the uptime of a server in a cluster, then environments that cause very frequent power failures render the server unusable, as the overhead of reconnecting it outweighs its usefulness. In such a case, being shut down is better than continuously trying to reconnect, and so we give a higher satisfaction value for the server being down, which depends only on the environment. Then, as demonstrated with the cleaning robot in Sect. 1, the difference and ratio functions are fairly natural when measuring "realization of potential". We now describe a more detailed example when these measures are in order.

Example 3. Consider a controller for an elevator in an n-floor building. The environment sends to the controller requests, by means of a truth assignment to $I = \{1, \ldots, n\}$, indicating the subset of floors in which the elevator is requested. Then, the controller assigns values to $O = \{up, down\}$, directing the elevator to go up, go down, or stay. The satisfaction value of the specification ψ reflects the waiting time of the request with the slowest response: it is 0 when this time is more than $2n$, and is 1 when the slowest request is granted immediately. Sure enough, there is no controller that attains satisfaction value 1 on all input sequences, and so ψ is not realizable with satisfaction value 1. Also, adding assumptions about the behavior of the environment is not of much interest. Using

AG GE-realizability, we can synthesize a controller that behaves in an optimal way. For example, using the difference function, we measure the performance of the controller on an input sequence $x \in (2^I)^\omega$ with respect to the best possible performance on x. Note that such a best performance needs a look-ahead on requests yet to come, which is indeed the satisfaction value of $\exists O.\psi$ in x. Thus, the assumption $[\![x, \exists O.\psi]\!]$ actually gives us the performance of a good-enough *off-line* controller. Accordingly, using the ratio function, we can synthesize a system with the best *competitive ratio* for an on-line interaction [7]. □

Given an LTL[$\mathcal{F}$] formula ψ and a function $\texttt{comb}$, we define the GE-*AG-realization value* of ψ in a transducer $\mathcal{T}$ by $\min\{\texttt{comb}([\![x, \exists O.\psi]\!], [\![x \otimes \mathcal{T}(x), \psi]\!]) : x \in (2^I)^\omega\}$. Then, our goal in *AG* GE*-realizability* is to find, given an LTL[$\mathcal{F}$] formula ψ and a function $\texttt{comb}$, the maximal value $v \in [0,1]$ such that there exists a transducer $\mathcal{T}$ whose AG GE-realization value of ψ is v. The *AG* GE *-synthesis* problem is then to find such a transducer.

We start by solving the decision version of AG GE-realizability.

Theorem 7. *The problem of deciding, given an* LTL[$\mathcal{F}$] *formula ψ, a function* $\texttt{comb}$*, and a threshold $v \in [0,1]$, whether there exists a transducer $\mathcal{T}$ whose AG* GE*-realization value of ψ is v, is 2EXPTIME-complete.*

Proof. Recall that $V(\psi)$ is the set of possible satisfaction values of ψ (and hence of $\exists O.\psi$), and that by Theorem 4, we have that $|V(\psi)| \leq 2^{|\psi|}$. Let $G_v = \{\langle v_1, v_2\rangle \in V(\psi) \times V(\psi) : \texttt{comb}(v_1, v_2) \geq v\}$. Intuitively, G is the set of satisfaction-value pairs $\langle [\![w, \exists O.\psi]\!], [\![w, \psi]\!]\rangle$ that are allowed to be generated by a transducer whose AG GE-realization value of ψ is at least v. By definition, AG GE-realization of ψ with value v coincides with realization of the language $L_v = \{w \in (2^{I\cup O})^\omega : \texttt{comb}([\![w, \exists O.\psi]\!], [\![w, \psi]\!]) \geq v\}$. By the monotonicity assumption on $\texttt{comb}$, for every $\langle v_1, v_2\rangle \in G_v$, we have that $\langle v_1', v_2'\rangle \in G$ for every $v_1' \leq v_1$ and $v_2' \geq v_2$. Hence, we can write $L_v = \bigcup_{\langle v_1, v_2\rangle \in G_v} \{w \in (2^{I\cup O})^\omega : [\![w, \exists O.\psi]\!] \leq v_1 \text{ and } [\![w, \psi]\!] \geq v_2\}$, and proceed to construct an NBW for L_v by taking the union of NBWs $\mathcal{A}_{v_1,v_2}$ for all $\langle v_1, v_2\rangle \in G_v$, each of which is the product of NBWs $\mathcal{A}^{\leq v_1}_{\exists O.\psi}$ and $\mathcal{A}^{\geq v_2}_{\psi}$, as in the proof of Theorem 5.

Aiming to proceed Safralessly, we can also construct a UCW for L_v, as follows. First, note that by the monotonicity of $\texttt{comb}$, for every $\langle v_1, v_2\rangle \in V(\psi) \times V(\psi)$ we have that $\langle v_1, v_2\rangle \in G_v$ iff for every $\langle u_1, u_2\rangle \in V(\psi) \times V(\psi) \setminus G_v$, we have that $v_1 < u_1$ or $v_2 > u_2$. Hence, $L_v = \bigcap_{\langle u_1, u_2\rangle \in V(\psi)\times V(\psi)\setminus G_v} \{w \in (2^{I\cup O})^\omega : [\![w, \exists O.\psi]\!] < u_1 \text{ or } [\![w, \psi]\!] > u_2\}$, and so by dualization we have $(2^{I\cup O})^\omega \setminus L_v = \bigcup_{\langle u_1, u_2\rangle \in V(\psi)\times V(\psi)\setminus G_v} \{w \in (2^{I\cup O})^\omega : [\![w, \exists O.\psi]\!] \geq u_1 \text{ and } [\![w, \psi]\!] \leq u_2\}$. Hence, we can obtain a UCW for L_v by dualizing an NBW that is the union of NBWs $\mathcal{A}_{u_1,u_2}$, for all $\langle u_1, u_2\rangle \in V(\psi) \times V(\psi) \setminus G_v$, each of which is the product of NBWs $\mathcal{A}^{\geq u_1}_{\exists O.\psi}$ and $\mathcal{A}^{\leq u_2}_{\psi}$.

Observe that in all cases, the size of the NBW is $2^{O(|\psi|)}$. Indeed, there are at most $2^{2|\psi|}$ pairs in the union, and, by Theorem 4, the size of the NBW for each pair is $2^{O(|\psi|)}$.

The lower bound follows from the 2EXPTIME-hardness of LTL GE-realizability. □

By Theorem 4, the number of possible satisfaction values for ψ is at most $2^{|\psi|}$. Thus, the number of possible values for $\texttt{comb}(A, G)$, where A and G are satisfaction values of ψ, is at most $2^{2|\psi|}$. Using binary search over the image of comb, we can use Theorem 7 to obtain the following.

Corollary 1. *The AG* GE*-synthesis problem can be solved in doubly-exponential time.*

Remark 5 [GE**-synthesis as a special case of AG** GE**-synthesis].** The two approaches taken in Sect. 5.2 can be captured by an appropriate comb function. Indeed, for GE-synthesis with a threshold, we can use the function comb with $\texttt{comb}(A, G) = 1$ if $A \geq v \rightarrow G \geq v$, and $\texttt{comb}(A, G) = 0$ otherwise. For GE-synthesis (without a threshold), we can use the function comb with $\texttt{comb}(A, G) = 1$ if $A = G$, and $\texttt{comb}(A, G) = 0$ otherwise (recall that $A \geq G$ by definition). However, the solution described in Sect. 5.2 is simpler than the one described here for the general case. □

5.4 LTL[$\mathcal{F}$] GE-Synthesis in Stochastic Environments

The setting of LTL[$\mathcal{F}$] GE-synthesis studied in Sects. 5.2 and 5.3 takes the different satisfaction values into an account, but is binary, in the sense that a specification is either (possibly AG) GE-realizable, or is not. In particular, in case the specification is not GE-realizable, synthesis algorithms only return "no". In this section we add a quantitative measure also to the underlying realizability question. We do so by assuming a stochastic environment, with a known distribution on the inputs sequences, and analyzing the expected performance of the system.

For completeness, we remind the reader of some basics of probability theory. For a comprehensive reference see e.g., [25]. Let Σ be a finite alphabet, and let ν be some *probability distribution* over Σ^ω. For example, in the uniform distribution over $(2^I)^\omega$, the probability space is induced by sampling each letter with probability $2^{-|I|}$, corresponding to settings in which each signal in I always holds in probability $\frac{1}{2}$. We assume ν is given by a finite Markov Decision Process (MDP). That is, ν is induced by the distribution of each letter $i \in 2^I$ at each time step, determined by a finite stochastic control process that takes into account also the outputs generated by the system (see [2] for the precise model). A *random variable* is then a function $X : \Sigma^\omega \rightarrow \mathbb{R}$. When X has a finite image V, which is the case in our setting, its *expected value* is $\mathbb{E}[X] = \sum_{v \in V} v \cdot \Pr(X^{-1}(v))$. Intuitively, $\mathbb{E}[X]$ is the "average" value that X attains. Next, consider an *event* $E \subseteq \Sigma^\omega$. The *conditional expectation of* X *with respect to* E is $\mathbb{E}[X|E] = \frac{\mathbb{E}[\mathbb{1}_E X]}{\Pr(E)}$, where $\mathbb{1}_E X$ is the random variable that assigns $X(w)$ to $w \in E$ and 0 to $w \notin E$. Intuitively, $\mathbb{E}[X|E]$ is the average value that X attains when restricting to words in E, and normalizing according to the probability of E itself.

We continue and review the *high-quality synthesis problem* [2], where the GE variant is not considered. There, the environment is assumed to be stochastic and we care for the expected satisfaction value of an LTL[$\mathcal{F}$] specification in

the computations of a transducer $\mathcal{T}$, assuming some given distribution on the inputs sequences. Formally, let $X_{\mathcal{T},\psi} : (2^I)^\omega \to \mathbb{R}$ be a random variable that assigns each sequence $x \in (2^I)^\omega$ of input signals with $[\![\mathcal{T}(x), \psi]\!]$. Then, when the sequences in $(2^I)^\omega$ are sampled according to a given distribution ν of $(2^I)^\omega$, we define $[\![\mathcal{T}, \psi]\!]^\nu = \mathbb{E}[X_{\mathcal{T},\psi}]$. Since ν is fixed, we omit it from the notation and use $[\![\mathcal{T}, \psi]\!]$ in the following.

Remark 6 **[Relating LTL GE-synthesis with stochastic LTL[$\mathcal{F}$] synthesis]** Given an LTL formula ψ, we can view it as an LTL[$\mathcal{F}$] formula with possible satisfaction values $\{0, 1\}$, apply to it high-quality synthesis *a-la* [2], and find a transducer $\mathcal{T}$ that maximizes $\mathbb{E}[X_{\mathcal{T},\psi}]$. An interesting observation is that if $\mathcal{T}$ GE-realizes ψ, then it also maximizes $\mathbb{E}[X_{\mathcal{T},\psi}]$. Indeed, all input sequences that can contribute to the expected satisfaction value, do so. □

We introduce and study two measures for high-quality synthesis in a stochastic environment. In the first, termed *expected* GE*-synthesis*, all input sequences are sampled, yet the satisfaction value in each input sequence takes its hopefulness level into account. In the second, termed *conditional expected* GE*-synthesis*, only hopeful input sequences are sampled.

We start with expected GE-synthesis. There, instead of associating each sequence $x \in (2^I)^\omega$ with $[\![x \otimes \mathcal{T}(x), \psi]\!]$, we associate it with $X^{\mathtt{comb}}_{\mathcal{T},\psi} = \mathtt{comb}([\![x, \exists O.\psi]\!], [\![x \otimes \mathcal{T}(x), \psi]\!])$, where `comb` is as described in Sect. 5.3, thus capturing the assume-guarantee semantics of quantitative GE-synthesis. Then, we define $[\![\mathcal{T}, \psi]\!]^{\mathtt{comb}} = \mathbb{E}[X^{\mathtt{comb}}_{\mathcal{T},\psi}]$. For example, taking `comb` as implication, we have $X^{\mathtt{comb}}_{\mathcal{T},\psi} = \max\{[\![x \otimes \mathcal{T}(x), \psi]\!], [\![x, \forall O.\neg\psi]\!]\}$, capturing the semantics of $(\exists O.\psi) \to \psi$.

Then, in conditional expected GE-synthesis, we consider $\exists O.\psi$ as an environment assumption, and factor it in using conditional expectation, parameterized by a threshold $v \in [0, 1]$. Formally, let $\exists O.\psi \geq v$ denote the event $\{x \in (2^I)^\omega : [\![x, \exists O.\psi]\!] \geq v\}$. Then, we define $[\![\mathcal{T}, \psi]\!]^{\mathrm{cond}(v)} = \mathbb{E}[X_{\mathcal{T},\psi} | \exists O.\psi \geq v]$, assuming the event $\exists O.\psi \geq v$ has a strictly positive probability.

In [2], it is shown that the high-quality synthesis problem can be solved in doubly-exponential time, also in the presence of environment assumptions. In the solution, the first step is the translation of the involved formulas to DPWs. In order to extract from [2] the results relevant to us, we describe them by means of *discrete quantitative specifications*, defined as follows. A discrete quantitative specification Ψ over $I \cup O$ is given by means of a sequence $\mathcal{A}_1, \ldots, \mathcal{A}_n$ of DPWs, with $(2^{I \cup O})^\omega = L(\mathcal{A}_1) \supseteq L(\mathcal{A}_2) \supseteq \ldots \supseteq L(\mathcal{A}_n)$, and sequence $0 \leq v_1 < \ldots < v_n \leq 1$ of values. For every $w \in (2^{I \cup O})^\omega$, the satisfaction value of w in Ψ, denoted $[\![w, \Psi]\!]$, is $\max\{v_i : w \in L(\mathcal{A}_i)\}$. We refer to n as the depth of Ψ.

Theorem 8 ([2]). *Consider a discrete quantitative specification Ψ over $I \cup O$. Let n be its depth and m be the size of the largest DPW in Ψ. For a transducer $\mathcal{T}$, let $X_{\mathcal{T}}$ be a random variable that assigns a word $x \in (2^I)^\omega$ with $[\![x \otimes \mathcal{T}(x), \Psi]\!]$.*

1. We can synthesize a transducer $\mathcal{T}$ that maximizes $\mathbb{E}[X_{\mathcal{T}}]$ in time m^n.

2. *Given a DPW $\mathcal{B}$ over 2^I such that $\Pr(L(\mathcal{B})) > 0$, we can synthesize a transducer $\mathcal{T}$ that maximizes $\mathbb{E}[X_{\mathcal{T}}|\mathcal{B}]$ in time $m^n \cdot k$, where k is the size of $\mathcal{B}$.*

We can now state the main results of this section.

Theorem 9. *Consider an* LTL$[\mathcal{F}]$ *formula ψ.*

1. *Given a function* comb*, we can find in doubly-exponential time a transducer that maximizes $[\![\mathcal{T}, \psi]\!]^{\mathtt{comb}}$.*
2. *Given a threshold $v \in [0, 1]$, we can find in doubly-exponential time a transducer that maximizes $[\![\mathcal{T}, \psi]\!]^{\mathrm{cond}(v)}$.*

Proof. Let $v_1 < v_2 < \ldots < v_n$ be the possible satisfaction values of ψ (and hence also of $\exists O.\psi$ and of $\forall O.\psi$). By Theorem 4, we have that $n \leq 2^{|\psi|}$. For each v_i, we can construct a DPW $\mathcal{D}^{\geq v_i}_{\mathtt{comb}(\exists O.\psi,\psi)}$ as in Theorem 7. It is not hard to see that the discrete quantitative specification given by the DPWs $\mathcal{D}^{\geq v_i}_{\mathtt{comb}(\exists O.\psi,\psi)}$ and the values v_i, for $1 \leq i \leq n$, is qual to the specification $\mathtt{comb}(\exists O.\psi, \psi)$. Thus, by Theorem 8 (1), we can find a transducer that maximizes $\mathbb{E}[X_{\mathcal{T}}]$ in time $(2^{2^{O(|\psi|)}})^{2^{|\psi|}} = 2^{2^{O(|\psi|)}}$.

Next, given $v \in [0, 1]$, we can check whether $\Pr(\exists O.\psi > v) > 0$, for example by converting a DPW $\mathcal{D}^{\geq v}_{\exists O.\psi}$ to an MDP, and reasoning about its Ergodic-components. Then, by Theorem 8 (2), we can find a transducer that maximizes $\mathbb{E}[X_{\mathcal{T}}|\exists O\psi > v]$, in time $(2^{2^{O(|\psi|)}})^{2^{|\psi|}} \cdot 2^{2^{O(\psi)}} = 2^{2^{O(|\psi|)}}$. □

Corollary 2. *The (possibly conditional) expected* GE*-synthesis problem for* LTL$[\mathcal{F}]$ *can be solved in doubly-exponential time.*

5.5 Guarantees in High-Quality GE-Synthesis

As in the Boolean setting, also in the high-quality one we would like to add to a GE-realizing transducer guarantees and indications about the satisfaction level. As we detail below, the quantitative setting offers many possible ways to do so.

High-Quality GE-Synthesis with Guarantees. We consider specifications of the form $\psi = \psi_{strong} \wedge \psi_{weak}$, where essentially, we seek a transducer that realizes ψ_{strong} and (possibly AG) GE-realizes ψ_{weak}. Maximizing the realization value of ψ_{strong} may conflict with maximizing the GE-realization value of ψ_{weak}, and there are different ways to trade-off the two goals. Technically, in the decision-problem variant, we are given two thresholds $v_1, v_2 \in [0, 1]$, and we seek a transducer $\mathcal{T}$ that realizes ψ_{strong} with value at least v_1, and GE-realizes ψ_{weak} with value at least v_2. Then, one may start, for example, by maximizing the value v_1, and then find the maximal value v_2 that may be achieved simultaneously. Alternatively, one may prefer to maximize v_2, or some other combination of v_1 and v_2. Also, it is possible to decompose ψ further, to several strong and weak components, each with its desired threshold.

The solutions in the different settings all involve a construction of a UCW $\mathcal{A}^{\geq v_1}_{\psi_{strong}}$, and its product with the automata constructed in the solutions for the

different GE-synthesis variants. We thus have the following. We note that when the solution for ψ_{weak} is Safraless, we can use a UCW for ψ_{strong} to maintain a Safraless construction.

Theorem 10. *The problem of* LTL[$\mathcal{F}$] *high-quality* GE*-synthesis with a guarantee can be solved in doubly-exponential time.*

Flags by a High-Quality GE-Realizing Transducer. In the quantitative setting, we parameterized the flags raised by the GE-realizing transducer by values in $[0,1]$, indicating the announced satisfaction level. Thus, rather than talking about prefixes being green, red, or blue, we talk about them being v-green, v-red, and v-blue, for $v \in [0,1]$, which essentially means that a satisfaction value of at least v is guarantees (in green and blue flags) or is impossible (in red ones). We can think of those as "degrees" of green, red, and blue. Below, we formalize this intuition and argue that even an augmentation of a transducer that GE-realizes ψ by flags for all values in $V(\psi)$ leaves the problem in doubly-exponential time.

A *quantitative language* over $2^{I \cup O}$ is $L : (2^{I \cup O})^\omega \to [0,1]$. For a quantitative language L and a word $w \in (2^{I \cup O})^*$, we define L^w as the quantitative language where for all $w' \in (2^{I \cup O})^\omega$, we have $L^w(w') = L(w \cdot w')$. For a value $v \in [0,1]$, a word $w \in (2^{I \cup O})^*$ is *v-green for L* if L^w is v-realizable. That is, there is a transducer $\mathcal{T}$ such that $[\![T, L^w]\!] \geq v$. A word $x \in (2^I)^*$ is *v-green for L* if there is $y \in (2^O)^*$ such that $x \otimes y$ is v-green for L. Thus, when the environment generates x, the system can respond in a way that would guarantee v-realizability. Finally, we say that L is *green realizable* if there is a strategy $f : (2^I)^+ \to 2^O$ that for every threshold v and for every input $x \in (2^I)^+$ that is v-green for L, we have that $x \otimes f(x)$ is v-green for L. It is not hard to see that Theorem 3 carries over to the quantitative setting, thus quantitative optimal realizability is strictly stronger than quantitative green realizability. In particular, if a transducer $\mathcal{T}$ optimally realizes an LTL[$\mathcal{F}$] formula ψ, then $\mathcal{T}$ also green realizes ψ. In the full version, we describe quantitative definitions also for red and blue prefixes, and describe monitors for the detection of the various types of prefixes.

6 Discussion

We introduced and solved several variants of GE-synthesis. Our complexity results are tight and show that GE-synthesis is not more complex than traditional synthesis. In practice, however, traditional synthesis algorithms do not scale well, and much research is devoted for the development of methods and heuristics for coping with the implementation challenges of synthesis. A natural future research direction is to extend these heuristics and methods for GE-synthesis. We mention here two specific examples.

Efficient synthesis algorithms have been developed for fragments of LTL [21]. Most notable is the *GR(1) fragment* [18], which supports assume-guarantee reasoning, and for which synthesis has an efficient symbolic solution. Adding existential quantification to GR(1) specifications, which is how we handled LTL

GE-synthesis, is not handled by its known algorithms, and is an interesting challenge. The success of SAT-based model-checking have led to the development of SAT-based synthesis algorithms [6], where the synthesis problem is reduced to satisfiability of a QBF formula. The fact the setting already includes quantifiers suggests it can be extended to GE-synthesis. A related effort is *bounded synthesis* algorithms [13,24], where the synthesized systems are assumed to be of a bounded size and can be represented symbolically [10].

References

1. Almagor, S., Boker, U., Kupferman, O.: Formalizing and reasoning about quality. J. ACM **63**(3), 24:1–24:56 (2016)
2. Almagor, S., Kupferman, O.: High-quality synthesis against stochastic environments. In: Proceedings of 25th Annual Conference of the European Association for Computer Science Logic, LIPIcs, vol. 62, pp. 28:1–28:17 (2016)
3. Almagor, S., Kupferman, O., Ringert, J.O., Velner, Y.: Quantitative assume guarantee synthesis. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 353–374. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_19
4. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_14
5. Bloem, R., Chatterjee, K., Jobstmann, B.: Graph games and reactive synthesis. Handbook of Model Checking, pp. 921–962. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_27
6. Bloem, R., Egly, U., Klampfl, P., Könighofer, R., Lonsing, F.: Sat-based methods for circuit synthesis. In: Proceedings of 14th International Conference on Formal Methods in Computer-Aided Design, pp. 31–34. IEEE (2014)
7. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press, New York (1998)
8. Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 147–161. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_14
9. Church, A.: Logic, arithmetics, and automata. In: Proceedings of International Congress of Mathematicians, vol. 1962, pp. 23–35. Institut Mittag-Leffler (1963)
10. Ehlers, R.: Symbolic bounded synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 365–379. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_33
11. Fisman, D., Kupferman, O., Lustig, Y.: Rational synthesis. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 190–204. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_16
12. Kupferman, O.: Automata theory and model checking. In: Clarke, E., Henzinger, T., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 107–151. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_4
13. Kupferman, O., Lustig, Y., Vardi, M.Y., Yannakakis, M.: Temporal synthesis for bounded systems and environments. In: Proceedings of 28th Symposium on Theoretical Aspects of Computer Science, pp. 615–626 (2011)

14. Kupferman, O., Perelli, G., Vardi, M.Y.: Synthesis with rational environments. Ann. Math. Artif. Intell. **78**(1), 3–20 (2016). https://doi.org/10.1007/s10472-016-9508-8
15. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_6
16. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Des. **19**(3), 291–314 (2001). https://doi.org/10.1023/A:1011254632723
17. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proceedings of 46th IEEE Symposium on Foundations of Computer Science, pp. 531–540 (2005)
18. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_24
19. Pnueli, A.: The temporal semantics of concurrent programs. Theor. Comput. Sci. **13**, 45–60 (1981)
20. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of 16th ACM Symposium on Principles of Programming Languages, pp. 179–190 (1989)
21. Alur, R., La Torre, S., Madhusudan, P.: Playing games with boxes and diamonds. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 128–143. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45187-7_8
22. Rosner, R.: Modular synthesis of reactive systems. Ph.D thesis, Weizmann Institute of Science (1992)
23. Safra, S.: On the complexity of ω-automata. In: Proceedings of 29th IEEE Symposium on Foundations of Computer Science, pp. 319–327 (1988)
24. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75596-8_33
25. Sheldon, R.: A First Course in Probability. Pearson Education India, Delhi (2002)
26. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with applications to temporal logic. Theor. Comput. Sci. **49**, 217–237 (1987)
27. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Inf. Comput. **115**(1), 1–37 (1994)
28. Winnicott, D.W.: Playing and Reality. Penguin, Harmondsworth (1971)
29. Wolper, P.: Temporal logic can be more expressive. In: Proceedings of 22nd IEEE Symposium on Foundations of Computer Science, pp. 340–348 (1981)

Synthesizing JIT Compilers for In-Kernel DSLs

Jacob Van Geffen[1(✉)], Luke Nelson[1], Isil Dillig[2], Xi Wang[1], and Emina Torlak[1]

[1] University of Washington, Seattle, USA
jsvg@cs.washington.edu
[2] University of Texas at Austin, Austin, USA

Abstract. Modern operating systems allow user-space applications to submit code for kernel execution through the use of in-kernel domain specific languages (DSLs). Applications use these DSLs to customize system policies and add new functionality. For performance, the kernel executes them via just-in-time (JIT) compilation. The correctness of these JITs is crucial for the security of the kernel: bugs in in-kernel JITs have led to numerous critical issues and patches.

This paper presents JitSynth, the first tool for synthesizing verified JITs for in-kernel DSLs. JitSynth takes as input interpreters for the source DSL and the target instruction set architecture. Given these interpreters, and a mapping from source to target states, JitSynth synthesizes a verified JIT compiler from the source to the target. Our key idea is to formulate this synthesis problem as one of synthesizing a per-instruction compiler for *abstract register machines*. Our core technical contribution is a new *compiler metasketch* that enables JitSynth to efficiently explore the resulting synthesis search space. To evaluate JitSynth, we use it to synthesize a JIT from eBPF to RISC-V and compare to a recently developed Linux JIT. The synthesized JIT avoids all known bugs in the Linux JIT, with an average slowdown of 1.82× in the performance of the generated code. We also use JitSynth to synthesize JITs for two additional source-target pairs. The results show that JitSynth offers a promising new way to develop verified JITs for in-kernel DSLs.

Keywords: Synthesis · Just-in-time compilation · Symbolic execution

1 Introduction

Modern operating systems (OSes) can be customized with user-specified programs that implement functionality like system call whitelisting, performance profiling, and power management [11,12,24]. For portability and safety, these programs are written in restricted domain-specific languages (DSLs), and the kernel executes them via interpretation and, for better performance, just-in-time (JIT) compilation. The correctness of in-kernel interpreters and JITs is crucial for the reliability and security of the kernel, and bugs in their implementations

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 564–586, 2020.
https://doi.org/10.1007/978-3-030-53291-8_29

have led to numerous critical issues and patches [15,30]. More broadly, embedded DSLs are also used to customize—and compromise [6,18]—other low-level software, such as font rendering and anti-virus engines [8]. Providing formal guarantees of correctness for in-kernel DSLs is thus a pressing practical and research problem with applications to a wide range of systems software.

Prior work has tackled this problem through interactive theorem proving. For example, the Jitk framework [40] uses the Coq interactive theorem prover [38] to implement and verify the correctness of a JIT compiler for the classic Berkeley Packet Filter (BPF) language [24] in the Linux kernel. But such an approach presents two key challenges. First, Jitk imposes a significant burden on DSL developers, requiring them to implement both the interpreter and the JIT compiler in Coq, and then manually prove the correctness of the JIT compiler with respect to the interpreter. Second, the resulting JIT implementation is extracted from Coq into OCaml and cannot be run in the kernel; rather, it must be run in user space, sacrificing performance and enlarging the trusted computing base (TCB) by relying on the OCaml runtime as part of the TCB.

This paper addresses these challenges with JitSynth, the first tool for synthesizing verified JIT compilers for in-kernel DSLs. JitSynth takes as input interpreters for the source DSL and the target instruction set architecture (ISA), and it synthesizes a JIT compiler that is guaranteed to transform each source program into a semantically equivalent target program. Using JitSynth, DSL developers write no proofs or compilers. Instead, they write the semantics of the source and target languages in the form of interpreters and a mapping from source to target states, which JitSynth trusts to be correct. The synthesized JIT compiler is implemented in C; thus, it can run directly in the kernel.

At first glance, synthesizing a JIT compiler seems intractable. Even the simplest compiler contains thousands of instructions, whereas existing synthesis techniques scale to tens of instructions. To tackle this problem in our setting, we observe that in-kernel DSLs are similar to ISAs: both take the form of bytecode instructions for an *abstract register machine*, a simple virtual machine with a program counter, a few registers, and limited memory store [40]. We also observe that in practice, the target machine has at least as many resources (registers and memory) as the source machine; and that JIT compilers for such abstract register machines perform register allocation statically at compile time. Our main insight is that we can exploit these properties to make synthesis tractable through *decomposition* and *prioritization*, while preserving soundness and completeness.

JitSynth works by decomposing the JIT synthesis problem into the problem of synthesizing individual *mini compilers* for every instruction in the source language. Each mini compiler is synthesized by generating a *compiler metasketch* [7], a set of ordered sketches that collectively represent *all* instruction sequences in the target ISA. These sketches are then solved by an off-the-shelf synthesis tool based on reduction to SMT [39]. The synthesis tool ensures that the target instruction sequence is semantically equivalent to the source instruction, according to the input interpreters. The order in which the sketches are explored is key to making this search practical, and JitSynth contributes two techniques for biasing the search towards tightly constrained, and therefore tractable, sketches that are likely to contain a correct program.

First, we observe that source instructions can often be implemented with target instructions that access the same parts of the state (e.g., only registers). Based on this observation, we develop *read-write sketches*, which restrict the synthesis search space to a subset of the target instructions, based on a sound and precise summary of their semantics. Second, we observe that hand-written JITs rely on pseudoinstructions to generate common target sequences, such as loading immediate (constant) values into registers. We use this observation to develop *pre-load sketches*, which employ synthesized pseudoinstructions to eliminate the need to repeatedly search for common target instruction subsequences.

We have implemented JitSynth in Rosette [39] and used it to synthesize JIT compilers for three widely used in-kernel DSLs. As our main case study, we used JitSynth to synthesize a RISC-V [32] compiler for extended BPF (eBPF) [12], an extension of classic BPF [24], used by the Linux kernel. Concurrently with our work, Linux developers manually built a JIT compiler for the same source and target pair, and a team of researchers found nine correctness bugs in that compiler shortly after its release [28]. In contrast, our JIT compiler is verified by construction; it supports 87 out of 102 eBPF instructions and passes all the Linux kernel tests within this subset, including the regression tests for these nine bugs. Our synthesized compiler generates code that is 5.24× faster than interpreted code and 1.82× times slower than the code generated by the Linux JIT. We also used JitSynth to synthesize a JIT from libseccomp [10], a policy language for system call whitelisting, to eBPF, and a JIT from classic BPF to eBPF. The synthesized JITs avoid previously found bugs in the existing generators for these source target pairs, while incurring, on average, a 2.28–2.61× slowdown in the performance of the generated code.

To summarize, this paper makes the following contributions:

1. JitSynth, the first tool for synthesizing verified JIT compilers for in-kernel DSLs, given the semantics of the source and target languages as interpreters.
2. A novel formulation of the JIT synthesis problem as one of synthesizing a per-instruction compiler for *abstract register machines*.
3. A novel *compiler metasketch* that enables JitSynth to solve the JIT synthesis problem with an off-the-shelf synthesis engine.
4. An evaluation of JitSynth's effectiveness, showing that it can synthesize verified JIT compilers for three widely used in-kernel DSLs.

The rest of this paper is organized as follows. Section 2 illustrates JitSynth on a small example. Section 3 formalizes the JIT synthesis problem for in-kernel DSLs. Section 4 presents the JitSynth algorithm for generating and solving compiler metasketches. Section 5 provides implementation details. Section 6 evaluates JitSynth. Section 7 discusses related work. Section 8 concludes.

2 Overview

This section provides an overview of JitSynth by illustrating how it synthesizes a toy JIT compiler (Fig. 1). The source language of the JIT is a tiny subset of

instruction	description	semantics
eBPF (subset):		
`addi32` *dst*, *imm32*	32-bit add (high 32 bits cleared)	$R[dst] \leftarrow 0^{32} \oplus (\text{extract}(31, 0, R[dst]) + imm32)$
RISC-V (subset):		
`lui` *rd*, *imm20*	load upper immediate	$R[rd] \leftarrow \text{sext64}(imm20 \oplus 0^{12})$
`addiw` *rd*, *rs*, *imm12*	32-bit register-immediate add	$R[rd] \leftarrow \text{sext64}(\text{extract}(31, 0, R[rs]) + \text{sext32}(imm12))$
`add` *rd*, *rs1*, *rs2*	register-register add	$R[rd] \leftarrow R[rs1] + R[rs2]$
`slli` *rd*, *rs*, *imm6*	register-immediate left shift	$R[rd] \leftarrow rs << (0^{58} \oplus imm6)$
`srli` *rd*, *rs*, *imm6*	register-immediate logical right shift	$R[rd] \leftarrow rs >> (0^{58} \oplus imm6)$
`lb` *rd*, *rs*, *imm12*	load byte from memory	$R[rd] \leftarrow \text{sext64}(M[R[rs] + \text{sext64}(imm12)])$
`sb` *rs1*, *rs2*, *imm12*	store byte to memory	$M[R[rs1] + \text{sext64}(imm12)] \leftarrow \text{extract}(7, 0, R[rs2])$

Fig. 1. Subsets of eBPF and RISC-V used as source and target languages, respectively, in our running example: $R[r]$ denotes the value of register r; $M[a]$ denotes the value at memory address a; $\oplus$ denotes concatenation of bitvectors; superscripts (e.g., 0^{32}) denote repetition of bits; $sext32(x)$ and $sext64(x)$ sign-extend x to 32 and 64 bits, respectively; and $\text{extract}(i, j, x)$ produces a subrange of bits of x from index i down to j.

eBPF [12] consisting of one instruction, and the target language is a subset of 64-bit RISC-V [32] consisting of seven instructions. Despite the simplicity of our languages, the Linux kernel JIT used to produce incorrect code for this eBPF instruction [27]; such miscompilation bugs not only lead to correctness issues, but also enable adversaries to compromise the OS kernel by crafting malicious eBPF programs [40]. This section shows how JitSynth can be used to synthesize a JIT that is verified with respect to the semantics of the source and target languages.

In-Kernel Languages. JitSynth expects the source and target languages to be a set of instructions for manipulating the state of an *abstract register machine* (Sect. 3). This state consists of a program counter (*pc*), a finite sequence of general-purpose registers (*reg*), and a finite sequence of memory locations (*mem*), all of which store bitvectors (i.e., finite precision integers). The length of these bitvectors is defined by the language; for example, both eBPF and RISC-V store 64-bit values in their registers. An instruction consists of an *opcode* and a finite set of *fields*, which are bitvectors representing either register identifiers or immediate (constant) values. For instance, the `addi32` instruction in eBPF has two fields: *dst* is a 4-bit value representing the index of the output register, and *imm32* is a 32-bit immediate. (eBPF instructions may have two additional fields *src* and *off*, which are not shown here as they are not used by `addi32`). An abstract register machine for a language gives meaning to its instructions: the machine consumes an instruction and a state, and produces a state that is the result of executing that instruction. Figure 1 shows a high-level description of the abstract register machines for our languages.

JitSynth *Interface.* To synthesize a compiler from one language to another, JitSynth takes as input their syntax, semantics, and a mapping from source to target states. All three inputs are given as a program in a *solver-aided host language* [39]. JitSynth uses Rosette as its host, but the host can be any language with a symbolic evaluation engine that can reduce the semantics of host

programs to SMT constraints (e.g., [37]). Figure 2 shows the interpreters for the source and target languages (i.e., emulators for their abstract register machines), as well as the state-mapping functions `regST`, `pcST`, and `memST` that JitSynth uses to determine whether a source state σ_S is equivalent to a target state σ_T. In particular, JitSynth deems these states equivalent, denoted by $\sigma_S \cong \sigma_T$, whenever $reg(\sigma_T)[\texttt{regST}(r)] = reg(\sigma_S)[r]$, $pc(\sigma_T) = \texttt{pcST}(pc(\sigma_S))$, and $mem(\sigma_T)[\texttt{memST}(a)] = mem(\sigma_S)[a]$ for all registers r and memory addresses a.

```
(struct state (regs mem pc) #:transparent)          ; Abstract register machine state.
                                                    ; Input 1/3: toy eBPF.
(struct ebpf-insn (opcode dst src off imm))         ;  - eBPF instruction format;
(define (ebpf-interpret insn st)                    ;  - eBPF interpreter for addi32.
  (define-match (ebpf-insn op dst _ _ imm) insn)    ;
  (case op                                          ; Note: addi32 does not use the src
    [(addi32)                                       ; and off fields.
     (state
      (reg-set st dst (concat (bv 0 32) (bvadd (extract 31 0 (reg-ref st dst)) imm)))
      (state-mem st)
      (bvadd (state-pc st) (bv 1 64)))]))
                                                    ; Input 2/3: toy RISC-V.
(struct rv-insn (opcode rd rs1 rs2 imm))            ;  - RISC-V instruction format;
(define (rv-interpret insn st)                      ;  - RISC-V interpreter.
  (define-match (rv-insn op rd rs1 rs2 imm) insn)
  (case op
    [(lui)
     (state
      (reg-set st rd (sext64 (concat imm (bv 0 12))))
      (state-mem st)
      (bvadd (state-pc st) (bv 4 64)))] ...))
                                                    ; Input 3/3: state mapping functions.
(define (regST r)                                   ;  - Register mapping:
  (cond [(equal? r (bv 0 4)) (bv 15 5)] ...))       ;    - eBPF r0 -> RISC-V x15, ...;
(define (memST a) a)                                ;  - Memory mapping is the identity.
(define (pcST pc) (bvshl pc (bv 2 64)))             ;  - PC mapping.
```

Fig. 2. Snippets of inputs to JitSynth: the interpreters for the source (eBPF) and and target (RISC-V) languages and state-mapping functions.

Decomposition into Per-instruction Compilers. Given these inputs, JitSynth generates a *per-instruction compiler* from the source to the target language. To ensure that the resulting compiler is correct (Theorem 1), and that one will be found if it exists (Theorem 2), JitSynth puts two restrictions on its inputs. First, the inputs must be self-finitizing [39], meaning that both the interpreters and the mapping functions must have a finite symbolic execution tree when applied to symbolic inputs. Second, the target machine must have at least as many registers and memory locations as the source machine; these storage cells must be as wide as those of the source machine; and the state-mapping functions (`pcST`, `regST`, and `memST`) must be injective. Our toy inputs satisfy these restrictions, as do the real in-kernel languages evaluated in Sect. 6.

Synthesis Workflow. JitSynth generates a per-instruction compiler for a given source and target pair in two stages. The first stage uses an optimized *compiler metasketch* to synthesize a mini compiler from every instruction in the source language to a sequence of instructions in the target language (Sect. 4).

The second stage then simply stitches these mini compilers into a full C compiler using a trusted outer loop and a switch statement. The first stage is a core technical contribution of this paper, and we illustrate it next on our toy example.

Metasketches. To understand how JitSynth works, consider the basic problem of determining if every `addi32` instruction can be emulated by a sequence of k instructions in toy RISC-V. In particular, we are interested in finding a program $C_{\texttt{addi32}}$ in our host language (which JitSynth translates to C) that takes as input a source instruction s = `addi32` *dst*, *imm32* and outputs a semantically equivalent RISC-V program $t = [t_1, \ldots, t_k]$. That is, for all *dst*, *imm32*, and for all equivalent states $\sigma_S \cong \sigma_T$, we have $run(s, \sigma_S, \texttt{ebpf-interpret}) \cong run(t, \sigma_T, \texttt{rv-interpret})$, where $run(e, \sigma, f)$ executes the instruction interpreter f on the sequence of instructions e, starting from the state σ (Definition 3).

We can solve this problem by asking the host synthesizer to search for $C_{\texttt{addi32}}$ in a space of candidate mini compilers of length k. We describe this space with a syntactic template, or a *sketch*, as shown below:

```
(define (compile-addi32 s)        ; Returns a list of k instruction holes, to be
  (define dst (ebpf-insn-dst s)) ; filled with toy RISC-V instructions. Each
  (define imm (ebpf-insn-imm s)) ; hole represents a set of choices, defined
  (list (??insn dst imm) ...))    ; by the ??insn procedure.
(define (??insn . sf)             ; Takes as input source instruction fields and
  (define rd  (??reg sf))         ; uses them to construct target field holes.
  (define rs1 (??reg sf))         ; ??reg and ??imm field holes are bitvector
  (define rs2 (??reg sf))         ; expressions over sf and arbitrary constants.
  (choose*                        ; Returns an expression that chooses among
    (rv-insn lui rd rs1 rs2 (??imm 20 sf))   ; lui, addiw,
    ...                                      ; ..., and
    (rv-insn sb  rd rs1 rs2 (??imm 12 sf)))) ; sb instructions.
```

Here, `(??insn dst imm)` stands for a missing expression—a hole—that the synthesizer needs to fill with an instruction from the toy RISC-V language. To fill an instruction hole, the synthesizer must find an expression that computes the value of the target instruction's fields. JitSynth limits this expression language to bitvector expressions (of any depth) over the fields of the source instruction and arbitrary bitvector constants.

Given this sketch, and our correctness specification for $C_{\texttt{addi32}}$, the synthesizer will search the space defined by the sketch for a program that satisfies the specification. Below is an example of the resulting toy compiler from eBPF to RISC-V, synthesized and translated to C by JitSynth (without the outer loop):

```
void compile(struct bpf_insn *insn, struct rv_insn *tgt_prog) {
  switch (insn->op) {
  case BPF_ADDI32:
    tgt_prog[0] = /* lui   x6, extract(19, 0, (imm + 0x800) >> 12) */
      rv_lui(6, extract(19, 0, (insn->imm + 0x800) >> 12));
    tgt_prog[1] = /* addiw x6, x6, extract(11, 0, imm) */
      rv_addiw(6, 6, extract(11, 0, insn->imm));
    tgt_prog[2] = /* add   rd, rd, x6 */
      rv_add(regmap(insn->dst), regmap(insn->dst), 6);
    tgt_prog[3] = /* slli  rd, rd, 32 */
      rv_slli(regmap(insn->dst), regmap(insn->dst), 32);
    tgt_prog[4] = /* srli  rd, rd, 32 */
      rv_srli(regmap(insn->dst), regmap(insn->dst), 32);
    break;
  }
}
```

Once we know how to synthesize a compiler of length k, we can easily extend this solution into a naive method for synthesizing a compiler of any length.

We simply enumerate sketches of increasing lengths, $k = 1, 2, 3, \ldots$, invoke the synthesizer on each generated sketch, and stop as soon as a solution is found (if ever). The resulting ordered set of sketches forms a metasketch [7]—i.e., a search space and a strategy for exploring it—that contains all candidate mini compilers (in a subset of the host language) from the source to the target language. This naive metasketch can be used to find a mini compiler for our toy example in 493 min. However, it fails to scale to real in-kernel DSLs (Sect. 6), motivating the need for JitSynth's optimized compiler metasketches.

Compiler Metasketches. JitSynth optimizes the naive metasketch by extending it with two kinds of more tightly constrained sketches, which are explored first. A constrained sketch of size k usually contains a correct solution of a given size if one exists, but if not, JitSynth will eventually explore the naive sketch of the same length, to maintain completeness. We give the intuition behind the two optimizations here, and present them in detail in Sect. 4.

First, we observe that practical source and target languages include similar kinds of instructions. For example, both eBPF and RISC-V include instructions for adding immediate values to registers. This similarity often makes it possible to emulate a source instruction with a sequence of target instructions that access the same part of the state (the program counter, registers, or memory) as the source instruction. For example, `addi32` reads and writes only registers, not memory, and it can be emulated with RISC-V instructions that also access only registers. To exploit this observation, we introduce *read-write sets*, which summarize, soundly and precisely, how an instruction accesses state. JitSynth uses these sets to define *read-write sketches* for a given source instruction, including only target instructions that access the state in the same way as the source instruction. For instance, a read-write sketch for `addi32` excludes both `lb` and `sb` instructions because they read and write memory as well as registers.

Second, we observe that hand-written JITs use pseudoinstructions to simplify their implementation of mini compilers. These are simply subroutines or macros for generating target sequences that implement common functionality. For example, the Linux JIT from eBPF to RISC-V includes a pseudoinstruction for loading 32-bit immediates into registers. JitSynth mimics the way hand-written JITs use pseudoinstructions with the help of *pre-load sketches*. These sketches first use a synthesized pseudoinstruction to create a sequence of concrete target instructions that load source immediates into scratch registers; then, they include a compute sequence comprised of read-write instruction holes. Applying these optimizations to our toy example, JitSynth finds a mini compiler for `addi32` in 5 s—a roughly 6000× speedup over the naive metasketch.

3 Problem Statement

This section formalizes the compiler synthesis problem for in-kernel DSLs. We focus on JIT compilers, which, for our purposes, means one-pass compilers [11]. To start, we define *abstract register machines* as a way to specify the syntax

and semantics of in-kernel languages. Next, we formulate our compiler synthesis problem as one of synthesizing a set of sound *mini compilers* from a single source instruction to a sequence of target instructions. Finally, we show that these mini compilers compose into a sound JIT compiler, which translates every source program into a semantically equivalent target program.

Abstract Register Machines. An abstract register machine (ARM) provides a simple interface for specifying the syntax and semantics of an in-kernel language. The syntax is given as a set of abstract instructions, and the semantics is given as a transition function over instructions and machine states.

An *abstract instruction* (Definition 1) defines the name (op) and type signature ($\mathcal{F}$) of an operation in the underlying language. For example, the abstract instruction ($addi32, r \mapsto Reg, imm32 \mapsto BV(32)$) specifies the name and signature of the `addi32` operation from the eBPF language (Fig. 1). Each abstract instruction represents the (finite) set of all *concrete instructions* that instantiate the abstract instruction's parameters with values of the right type. For example, `addi32` $0, 5$ is a concrete instantiation of the abstract instruction for `addi32`. In the rest of this paper, we will write "instruction" to mean a concrete instruction.

Definition 1 (Abstract and Concrete Instructions). *An* abstract instruction ι *is a pair* $(op, \mathcal{F})$ *where op is an opcode and* $\mathcal{F}$ *is a mapping from* fields *to their* types. *Field types include Reg, denoting register names, and* $BV(k)$, *denoting k-bit bitvector values. The abstract instruction* ι *represents all* concrete instructions $p = (op, F)$ *with the opcode op that bind each field* $f \in dom(\mathcal{F})$ *to a value* $F(f)$ *of type* $\mathcal{F}(f)$. *We write* $P(\iota)$ *to denote the set of all concrete instructions for* ι, *and we extend this notation to sets of abstract instructions in the usual way, i.e.,* $P(\mathcal{I}) = \bigcup_{\iota \in \mathcal{I}} P(\iota)$ *for the set* $\mathcal{I}$.

Instructions operate on machine *states* (Definition 2), and their semantics are given by the machine's *transition function* (Definition 3). A machine state consists of a program counter, a map from register names to register values, and a map from memory addresses to memory values. Each state component is either a bitvector or a map over bitvectors, making the set of all states of an ARM finite. The transition function of an ARM defines an interpreter for the ARM's language by specifying how to compute the output state for a given instruction and input state. We can apply this interpreter, together with the ARM's *fuel function*, to define an *execution* of the machine on a program and an initial state. The fuel function takes as input a sequence of instructions and returns a natural number that bounds the number of steps (i.e., state transitions) the machine can make to execute the given sequence. The inclusion of fuel models the requirement of in-kernel languages for all program executions to terminate [40]. It also enables us to use symbolic execution to soundly reduce the semantics of these languages to SMT constraints, in order to formulate the synthesis queries in Sect. 4.5.

Definition 2 (State). *A* state σ *is a tuple* (pc, reg, mem) *where pc is a value, reg is a function from register names to values, and mem is a function from memory addresses to values. Register names, memory addresses, and all values*

are finite-precision integers, or bitvectors. We write $|\sigma|$ *to denote the* size *of the state* σ*. The size* $|\sigma|$ *is defined to be the tuple* $(r, m, k_{pc}, k_{reg}, k_{mem})$*, where* r *is the number of registers in* σ*,* m *is the number of memory addresses, and* k_{pc}*,* k_{reg}*, and* k_{mem} *are the width of the bitvector values stored in the pc, reg, and mem, respectively. Two states have the same size if* $|\sigma_i| = |\sigma_j|$*; one state is smaller than another,* $|\sigma_i| \leq |\sigma_j|$*, if each element of* $|\sigma_i|$ *is less than or equal to the corresponding element of* $|\sigma_j|$*.*

Definition 3 (Abstract Register Machines and Executions). *An* abstract register machine $\mathcal{A}$ *is a tuple* $(\mathcal{I}, \Sigma, \mathcal{T}, \Phi)$ *where* $\mathcal{I}$ *is a set of abstract instructions,* Σ *is a set of states of the same size,* $\mathcal{T} : P(\mathcal{I}) \to \Sigma \to \Sigma$ *is a* transition function *from instructions and states to states, and* $\Phi : List(P(\mathcal{I})) \to \mathbb{N}$ *is a* fuel function *from sequences of instructions to natural numbers. Given a state* $\sigma_0 \in \Sigma$ *and a sequence of instructions* $\boldsymbol{p}$ *drawn from* $P(\mathcal{I})$*, we define the* execution *of* $\mathcal{A}$ *on* $\boldsymbol{p}$ *and* σ_0 *to be the result of applying* $\mathcal{T}$ *to* $\boldsymbol{p}$ *at most* $\Phi(\boldsymbol{p})$ *times. That is,* $\mathcal{A}(\boldsymbol{p}, \sigma_0) = run(\boldsymbol{p}, \sigma_0, \mathcal{T}, \Phi(\boldsymbol{p}))$*, where*

$$run(\boldsymbol{p}, \sigma, \mathcal{T}, k) = \begin{cases} \sigma, & \text{if } k = 0 \text{ or } pc(\sigma) \notin [0, |\boldsymbol{p}|) \\ run(\boldsymbol{p}, \mathcal{T}(\boldsymbol{p}[pc(\sigma)], \sigma), \mathcal{T}, k-1), & \text{otherwise.} \end{cases}$$

Synthesizing JIT Compilers for ARMs. Given a source and target ARM, our goal is to synthesize a one-pass JIT compiler that translates source programs to semantically equivalent target programs. To make synthesis tractable, we fix the structure of the JIT to consist of an outer loop and a switch statement that dispatches compilation tasks to a set of *mini compilers* (Definition 4). Our synthesis problem is therefore to find a sound mini compiler for each abstract instruction in the source machine (Definition 5).

Definition 4 (Mini Compiler). *Let* $\mathcal{A}_S = (\mathcal{I}_S, \Sigma_S, \mathcal{T}_S, \Phi_S)$ *and* $\mathcal{A}_T = (\mathcal{I}_T, \Sigma_T, \mathcal{T}_T, \Phi_T)$ *be two abstract register machines,* $\cong$ *an equivalence relation on their states* Σ_S *and* Σ_T*, and* $C : P(\iota) \to List(P(\mathcal{I}_T))$ *a function for some* $\iota \in \mathcal{I}_S$*. We say that* C *is a* sound mini compiler *for* ι *with respect to* $\cong$ *iff*

$$\forall \sigma_S \in \Sigma_S,\ \sigma_T \in \Sigma_T,\ p \in P(\iota).\ \sigma_S \cong \sigma_T \Rightarrow \mathcal{A}_S(p, \sigma_S) \cong \mathcal{A}_T(C(p), \sigma_T)$$

Definition 5 (Mini Compiler Synthesis). *Given two abstract register machines* $\mathcal{A}_S = (\mathcal{I}_S, \Sigma_S, \mathcal{T}_S, \Phi_S)$ *and* $\mathcal{A}_T = (\mathcal{I}_T, \Sigma_T, \mathcal{T}_T, \Phi_T)$*, as well as an equivalence relation* $\cong$ *on their states, the* mini compiler synthesis problem *is to generate a sound mini compiler* C_ι *for each* $\iota \in \mathcal{I}_S$ *with respect to* $\cong$*.*

The general version of our synthesis problem, defined above, uses an arbitrary equivalence relation $\cong$ between the states of the source and target machines to determine if a source and target program are semantically equivalent. JitSynth can, in principle, solve this problem with the naive metasketch described in Sect. 2. In practice, however, the naive metasketch scales poorly, even on small languages such as toy eBPF and RISC-V. So, in this paper, we focus on source

and target ARMs that satisfy an additional assumption on their state equivalence relation: it can be expressed in terms of injective mappings from source to target states (Definition 6). This restriction enables JitSynth to employ optimizations (such as pre-load sketches described in Sect. 4.4) that are crucial to scaling synthesis to real in-kernel languages.

Definition 6 (Injective State Equivalence Relation). *Let $\mathcal{A}_S$ and $\mathcal{A}_T$ be abstract register machines with states Σ_S and Σ_T such that $|\sigma_S| \leq |\sigma_T|$ for all $\sigma_S \in \Sigma_S$ and $\sigma_T \in \Sigma_T$. Let $\mathcal{M}$ be a* state mapping *$(\mathcal{M}_{pc}, \mathcal{M}_{reg}, \mathcal{M}_{mem})$ from Σ_S and Σ_T, where $\mathcal{M}_{pc}$ multiplies the program counter of the states in Σ_S by a constant factor, $\mathcal{M}_{reg}$ is an injective map from register names in Σ_S to those in Σ_T, and $\mathcal{M}_{mem}$ is an injective map from memory addresses in Σ_S to those in Σ_T. We say that two states $\sigma_S \in \Sigma_S$ and $\sigma_T \in \Sigma_T$ are equivalent according to $\mathcal{M}$, written $\sigma_S \cong_{\mathcal{M}} \sigma_T$, iff $\mathcal{M}_{pc}(pc(\sigma_S)) = pc(\sigma_T)$, $reg(\sigma_S)[r] = reg(\sigma_T)[\mathcal{M}_{reg}(r)]$ for all register names $r \in \mathrm{dom}(reg(\sigma_S))$, and $mem(\sigma_S)[a] = mem(\sigma_T)[\mathcal{M}_{mem}(a)]$ for all memory addresses $a \in \mathrm{dom}(mem(\sigma_S))$. The binary relation $\cong_{\mathcal{M}}$ is called an* injective state equivalence relation *on $\mathcal{A}_S$ and $\mathcal{A}_T$.*

Soundness of JIT Compilers for ARMs. Finally, we note that a JIT compiler composed from the synthesized mini compilers correctly translates every source program to an equivalent target program. We formulate and prove this theorem using the Lean theorem prover [25].

Theorem 1 (Soundness of JIT compilers). *Let $\mathcal{A}_S = (\mathcal{I}_S, \Sigma_S, \mathcal{T}_S, \Phi_S)$ and $\mathcal{A}_T = (\mathcal{I}_T, \Sigma_T, \mathcal{T}_T, \Phi_T)$ be abstract register machines, $\cong_{\mathcal{M}}$ an injective state equivalence relation on their states such that $M_{pc}(pc(\sigma_S)) = N_{pc}pc(\sigma_S)$, and $\{C_1, \ldots, C_{|\mathcal{I}_S|}\}$ a solution to the mini compiler synthesis problem for $\mathcal{A}_S$, $\mathcal{A}_T$, and $\cong_{\mathcal{M}}$ where $\forall s \in P(\iota).\ |C_i(s)| = N_{pc}$. Let $\mathcal{C} : P(\mathcal{I}_S) \to List(P(\mathcal{I}_T))$ be a function that maps concrete instructions $s \in P(\iota)$ to the compiler output $C_\iota(s)$ for $\iota \in \mathcal{I}_S$. If $\boldsymbol{s} = s_1, \ldots, s_n$ is a sequence of concrete instructions drawn from $\mathcal{I}_S$, and $\boldsymbol{t} = \mathcal{C}(s_1) \cdot \ldots \cdot \mathcal{C}(s_n)$ where $\cdot$ stands for sequence concatenation, then $\forall \sigma_S \in \Sigma_S, \sigma_T \in \Sigma_T.\ \sigma_S \cong_{\mathcal{M}} \sigma_T \Rightarrow \mathcal{A}_S(\boldsymbol{s}, \sigma_S) \cong_{\mathcal{M}} \mathcal{A}_T(\boldsymbol{t}, \sigma_T)$.*

4 Solving the Mini Compiler Synthesis Problem

This section presents our approach to solving the mini compiler synthesis problem defined in Sect. 3. We employ syntax-guided synthesis [37] to search for an implementation of a mini compiler in a space of candidate programs. Our core contribution is an effective way to structure this space using a *compiler metasketch*. This section presents our algorithm for generating compiler metasketches, describes its key subroutines and optimizations, and shows how to solve the resulting sketches with an off-the-shelf synthesis engine.

4.1 Generating Compiler Metasketches

JitSynth synthesizes mini compilers by generating and solving *metasketches* [7]. A metasketch describes a space of candidate programs using an ordered set of

syntactic templates or *sketches* [37]. These sketches take the form of programs with missing expressions or *holes*, where each hole describes a finite set of candidate completions. JitSynth sketches are expressed in a *host language* $\mathcal{H}$ that serves both as the implementation language for mini compilers and the specification language for ARMs. JitSynth expects the host to provide a synthesizer for completing sketches and a symbolic evaluator for reducing ARM semantics to SMT constraints. JitSynth uses these tools to generate optimized metasketches for mini compilers, which we call *compiler metasketches*.

Figure 3 shows our algorithm for generating compiler metasketches. The algorithm, CMS, takes as input an abstract source instruction ι for a source machine $\mathcal{A}_S$, a target machine $\mathcal{A}_T$, and a state mapping $\mathcal{M}$ from $\mathcal{A}_S$ to $\mathcal{A}_T$. Given these inputs, it lazily enumerates an infinite set of *compiler sketches* that collectively represent the space of all straight-line bitvector programs from $P(\iota)$ to $List(P(\mathcal{I}_T))$. In particular, each compiler sketch consists of k target *instruction holes*, constructed from field holes that denote bitvector expressions (over the fields of ι) of depth d or less. For each length k and depth d, the CMS loop generates three kinds of compiler sketches: the *pre-load*, the *read-write*, and the *naive* sketch. The naive sketch (Sect. 4.2) is the most general, consisting of all candidate mini compilers of length k and depth d. But it also scales poorly, so CMS first yields the pre-load (Sect. 4.4) and read-write (Sect. 4.3) sketches. As we will see later, these sketches describe a subset of the programs in the naive sketch, and they are designed to prioritize exploring small parts of the search space that are likely to contain a correct mini compiler for ι, if one exists.

```
1: function CMS(ι, A_S, A_T, M)                    ▷ ι ∈ I_S, A_S = (I_S, ...)
2:     for n ∈ ℤ⁺ do                               ▷ Lazily enumerates all compiler sketches
3:         for k ∈ [1, n], d = n − k do            ▷ of length k and depth d,
4:             yield PLD(k, d, ι, A_S, A_T, M)     ▷ yielding the pre-load sketch first,
5:             yield RW(k, d, ι, A_S, A_T, M)      ▷ read-write sketch next, and
6:             yield Naive(k, d, ι, A_S, A_T, M)   ▷ the most general sketch last.
```

Fig. 3. Compiler metasketch for the abstract source instruction ι, source machine $\mathcal{A}_S$, target machine $\mathcal{A}_T$, and state mapping $\mathcal{M}$ from $\mathcal{A}_S$ to $\mathcal{A}_T$.

4.2 Generating Naive Sketches

The most general sketch we consider, $\textsc{Naive}(k, d, \iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$, is shown in Fig. 4. This sketch consists of k instruction holes that can be filled with any instruction from $\mathcal{I}_T$. An instruction hole chooses between expressions of the form (op_T, H), where op_T is a target opcode, and H specifies the field holes for that opcode. Each field hole is a bitvector expression (of depth d) over the fields of the input source instruction and arbitrary bitvector constants. This lets target instructions use the immediates and registers (modulo $\mathcal{M}$) of the source instruction, as well as arbitrary constant values and register names. Letting field holes

include constant register names allows the synthesized mini compilers to use target registers unmapped by $\mathcal{M}$ as temporary, or scratch, storage. In essence, the naive sketch describes all straight-line compiler programs that can make free use of standard C arithmetic and bitwise operators, as well as scratch registers.

The space of such programs is intractably large, however, even for small inputs. For instance, it includes at least 2^{350} programs of length $k = 5$ and depth $d \leq 3$ for the toy example from Sect. 2. JitSynth therefore employs two effective heuristics to direct the exploration of this space toward the most promising candidates first, as defined by the read-write and pre-load sketches.

```
1:  function Naive(k, d, ι, A_S, A_T, M)                ▷ ι ∈ I_S, A_S = (I_S, ...)
2:      (op, F) ← ι, (I_T, ...) ← A_T                   ▷ Source instruction, target instructions.
3:      p ← FreshId()                                   ▷ Identifier for the compiler's input.
4:      body ← []                                       ▷ The body of the compiler is a sequence
5:      for 0 ≤ i < k do                                ▷ of k target instruction holes.
6:          I ← {}                                      ▷ The set I of choices for a target instruction hole
7:          for (op_T, F_T) ∈ I_T do                    ▷ includes all instructions from I_T.
8:              E ← {Expr(p.f, M) | f ∈ dom(F)}         ▷ Any source field can appear in
9:              H ← {f ↦ Field(F_T(f), d, E) | f ∈ dom(F_T)}   ▷ a target field hole, and
10:             I ← I ∪ {Expr((op_T, H), M)}            ▷ any constant register or value.
11:         body ← body · [Choose(I)]                   ▷ Append a hole over I to the body.
12:     return Expr((λp ∈ P(ι) . body), M)              ▷ A mini compiler sketch for ι.
```

Fig. 4. Naive sketch of length k and maximum depth d for ι, $\mathcal{A}_S$, $\mathcal{A}_T$, and $\mathcal{M}$. Here, *Expr* creates an expression in the host language, using $\mathcal{M}$ to map from source to target register names and memory addresses; $Choose(E)$ is a hole that chooses an expression from the set E; and $Field(\tau, d, E)$ is a hole for a bitvector expression of type τ and maximum depth d, constructed from arbitrary bitvector constants and expressions E.

4.3 Generating Read-Write Sketches

The read-write sketch, $\mathrm{RW}(k, d, \iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$, is based on the observation that many practical source and target languages provide similar functionality, so a source instruction ι can often be emulated with target instructions that access the same parts of the state as ι. For example, the `addi32` instruction from eBPF reads and writes only registers (not, e.g., memory), and it can be emulated with RISC-V instructions that also touch only registers (Sect. 2). Moreover, note that the semantics of `addi32` ignores the values of its *src* and *off* fields, and that the target RISC-V instructions do the same. Based on these observations, our optimized sketch for `addi32` would therefore consists of instruction holes that allow only register-register instructions, with field holes that exclude *src* and *off*. We first formalize this intuition with the notion of *read and write sets*, and then describe how JitSynth applies such sets to create RW sketches.

Read and Write Sets. Read and write sets provide a compact way to summarize the semantics of an abstract instruction ι. This summary consists of a set of *state labels*, where a state label is one of L_{reg}, L_{mem}, and L_{pc} (Definition 7). Each label in a summary set represents a state component (registers, memory, or the program counter) that a concrete instance of ι may read or write during some execution. We compute three such sets of labels for every ι: the read set $Read(\iota)$, the write set $Write(\iota)$, and the write set $Write(\iota, f)$ for each field f of ι. Figure 5 shows these sets for the toy eBPF and RISC-V instructions.

ι	$Read(\iota)$	$Write(\iota)$	$Write(\iota, field)$
`addi32`	$\{L_{reg}\}$	$\{L_{reg}\}$	*imm*: $\{L_{reg}\}$; *off*: $\emptyset$; *src*: $\emptyset$; *dst*: $\{L_{reg}\}$
`lui`	$\{L_{reg}\}$	$\{L_{reg}\}$	*rd*: $\{L_{reg}\}$; *imm20*: $\{L_{reg}\}$
`sb`	$\{L_{reg}\}$	$\{L_{mem}\}$	*rs1*: $\{L_{mem}\}$; *rs2*: $\{L_{mem}\}$; *imm12*: $\{L_{mem}\}$

Fig. 5. Read and write sets for the `addi32`, `lui`, and `sb` instructions from Fig. 1.

The read set $Read(\iota)$ specifies which components of the input state may affect the execution of ι (Definition 8). For example, if $Read(\iota)$ includes L_{reg}, then some concrete instance of ι produces different output states when executed on two input states that differ only in register values. The write set $Write(\iota)$ specifies which components of the output state may be affected by executing ι (Definition 9). In particular, if $Write(\iota)$ includes L_{reg} (or L_{mem}), then executing some concrete instance of ι on an input state produces an output state with different register (or memory) values. The inclusion of L_{pc} is based on a separate condition, designed to distinguish jump instructions from fall-through instructions. Both kinds of instructions change the program counter, but fall-through instructions always change it in the same way. So, $L_{pc} \in Write(\iota)$ if two instances of ι can write different values to the program counter. Finally, the field write set, $Write(\iota, f)$, specifies the parts of the output state are affected by the value of the field f; $L_n \in Write(\iota, f)$ means that two instances of ι that differ only in f can produce different outputs when applied to the same input state.

JitSynth computes all read and write sets from their definitions, by using the host symbolic evaluator to reduce the reasoning about instruction semantics to SMT queries. This reduction is possible because we assume that all ARM interpreters are self-finitizing, as discussed in Sect. 2.

Definition 7 (State Labels). *A* state label *is an identifier L_n where n is a state component, i.e., $n \in \{reg, mem, pc\}$. We write N for the set of all state components, and $\mathcal{L}$ for the set of all state labels. We also use state labels to access the corresponding state components: $L_n(\sigma) = n(\sigma)$ for all $n \in N$.*

Definition 8 (Read Set). *Let $\iota \in \mathcal{I}$ be an abstract instruction in $(\mathcal{I}, \Sigma, \mathcal{T}, \Phi)$. The* read set *of ι, $Read(\iota)$, is the set of all state labels $L_n \in \mathcal{L}$ such that $\exists p \in P(\iota). \exists L_w \in Write(\iota). \exists \sigma_a, \sigma_b \in \Sigma. (L_n(\sigma_a) \neq L_n(\sigma_b) \wedge (\bigwedge_{m \in N \setminus \{n\}} L_m(\sigma_a) = L_m(\sigma_b)) \wedge L_w(\mathcal{T}(p, \sigma_a)) \neq L_w(\mathcal{T}(p, \sigma_b))$.*

Definition 9 (Write Set). *Let $\iota \in \mathcal{I}$ be an abstract instruction in $(\mathcal{I}, \Sigma, \mathcal{T}, \Phi)$. The* write set *of ι, $Write(\iota)$, includes the state label $L_n \in \{L_{reg}, L_{mem}\}$ iff $\exists p \in P(\iota). \exists \sigma \in \Sigma. L_n(\sigma) \neq L_n(\mathcal{T}(p, \sigma))$, and it includes the state label L_{pc} iff $\exists p_a, p_b \in P(\iota). \exists \sigma \in \Sigma. L_{pc}(\mathcal{T}(p_a, \sigma)) \neq L_{pc}(\mathcal{T}(p_b, \sigma))$.*

Definition 10 (Field Write Set). *Let f be a field of an abstract instruction $\iota = (op, \mathcal{F})$ in $(\mathcal{I}, \Sigma, \mathcal{T}, \Phi)$. The* write set *of ι and f, $Write(\iota, f)$, includes the state label $L_n \in \mathcal{L}$ iff $\exists p_a, p_b \in P(\iota). \exists \sigma \in \Sigma. (p_a.f \neq p_b.f) \wedge (\bigwedge_{g \in dom(\mathcal{F}) \setminus \{f\}} p_a.g = p_b.g) \wedge L_n(\mathcal{T}(p_a, \sigma)) \neq L_n(\mathcal{T}(p_b, \sigma))$, where $p.f$ denotes $F(f)$ for $p = (op, F)$.*

Using Read and Write Sets. Given the read and write sets for a source instruction ι and target instructions $\mathcal{I}_T$, JitSynth generates the RW sketch of length k and depth d by modifying the Naive algorithm (Fig. 4) as follows. First, it restricts each target instruction hole (line 7) to choose an instruction $\iota_T \in \mathcal{I}_T$ with the same read and write sets as ι, i.e., $Read(\iota) = Read(\iota_T)$ and $Write(\iota) = Write(\iota_T)$. Second, it restricts the target field holes (line 9) to use the source fields with the matching field write set, i.e., the hole for a target field f_T uses the source field f when $Write(\iota_T, f_t) = Write(\iota, f)$. For example, given the sets from Fig. 5, the RW instruction holes for `addi32` exclude `sb` but include `lui`, and the field holes for `lui` use only the *dst* and *imm* source fields. More generally, the RW sketch for `addi32` consists of register-register instructions over *dst* and *imm*, as intended. This sketch includes 2^{290} programs of length $k = 5$ and depth $d \leq 3$, resulting in a 2^{60} fold reduction in the size of the search space compared to the Naive sketch of the same length and depth.

4.4 Generating Pre-load Sketches

The pre-load sketch, PLD $(k, d, \iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$, is based on the observation that hand-written JITs use macros or subroutines to generate frequently used target instruction sequences. For example, compiling a source instruction with immediate fields often involves loading the immediates into scratch registers, and hand-written JITs include a subroutine that generates the target instructions for performing these loads. The pre-load sketch shown in Fig. 6 mimics this structure.

In particular, PLD generates a sequence of m concrete instructions that load the (used) immediate fields of ι, followed by a sequence of $k - m$ instruction holes. The instruction holes can refer to both the source registers (if any) and the scratch registers (via the arbitrary bitvector constants included in the *Field* holes). The function $Load(Expr(p.f), \mathcal{A}_T, \mathcal{M})$ returns a sequence of target instructions that load the immediate $p.f$ into an unused scratch register. This function itself is synthesized by JitSynth using a variant of the RW sketch.

As an example, the pre-load sketch for `addi32` consists of two *Load* instructions (`lui` and `addiw` in the generated C code) and $k-2$ instruction holes. The holes choose among register-register instructions in toy RISC-V, and they can refer to the *dst* register of `addi32`, as well as any scratch register. The resulting sketch includes 2^{100} programs of length $k = 5$ and depth $d \leq 3$, providing a 2^{190} fold reduction in the size of the search space compared to the RW sketch.

```
1:  function PLD(k, d, ι, A_S, A_T, M)                    ▷ ι ∈ I_S, A_S = (I_S, ...)
2:      (op, F) ← ι, (I_T, ...) ← A_T                     ▷ Source instruction, target instructions.
3:      p ← FreshId()                                     ▷ Identifier for the compiler's input source instruction.
4:      body ← []                                         ▷ The body of the compiler is a sequence with 2 parts:
5:      imm ← {f | F(f) = BV(k) and Write(ι, f) ≠ ∅}      ▷ (1) Load each relevant
6:      for f ∈ imm do                                    ▷ source immediate into a free scratch register
7:          body ← body · Load(Expr(p.f), A_T, M)         ▷ using the load pseudoinstruction.
8:      m ← |body|                                        ▷ Let m be the length of the load sequence.
9:      if m ≥ k or m = 0 then return ⊥                   ▷ Return the empty sketch if m ∉ (0..k).
10:     for m ≤ i < k do                                  ▷ (2) Create k − m target instruction holes, where the set
11:         I ← {}                                        ▷ I of choices for a target instruction hole includes
12:         for ι_T ∈ I_T, ι_T = (op_T, F_T) do           ▷ all instructions from I_T that read-write
13:             rw_T ← Read(ι_T) × Write(ι_T)             ▷ the same state as ι or just registers.
14:             if rw_T = Read(ι) × Write(ι) or rw_T ⊆ {L_reg} × {L_reg} then
15:                 regs ← {f | F(f) = Reg and Write(ι, f) ≠ ∅}   ▷ Any relevant
16:                 E ← {Expr(p.f, M) | f ∈ regs}         ▷ source register can appear in
17:                 H ← {f ↦ Field(F_T(f), d, E) | f ∈ dom(F_T)}   ▷ a target field hole,
18:                 I ← I ∪ {Expr((op_T, H), M)}          ▷ and any constant register or value.
19:         body ← body · [Choose(I)]                     ▷ Append a hole over I to the body.
20:     return Expr((λp ∈ P(ι) . body), M)                ▷ A mini compiler sketch for ι.
```

Fig. 6. Pre-load sketch of length k and maximum depth d for ι, $\mathcal{A}_S$, $\mathcal{A}_T$, and $\mathcal{M}$. The $Load(E, \mathcal{A}_T, \mathcal{M})$ function returns a sequence of target instructions that load the immediate value described by the expression E into an unused scratch register; see Fig. 4 for descriptions of other helper functions.

4.5 Solving Compiler Metasketches

JitSynth solves the metasketch $\text{CMS}(\iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$ by applying the host synthesizer to each of the generated sketches in turn until a mini compiler is found. If no mini compiler exists in the search space, this synthesis process runs forever. To check if a sketch $\mathcal{S}$ contains a mini compiler, JitSynth would ideally ask the host synthesizer to solve the following query, derived from Definitions 4–6:

$$\exists C \in \mathcal{S}.\ \forall \sigma_S \in \Sigma_S,\ \sigma_T \in \Sigma_T,\ p \in P(\iota). \sigma_S \cong_{\mathcal{M}} \sigma_T \Rightarrow \mathcal{A}_S(p, \sigma_S) \cong_{\mathcal{M}} \mathcal{A}_T(C(p), \sigma_T)$$

But recall that the state equivalence check $\cong_{\mathcal{M}}$ involves universally quantified formulas over memory addresses and register names. In principle, these innermost quantifiers are not problematic because they range over finite domains (bitvectors) so the formula remains decidable. In practice, however, they lead to intractable SMT queries. We therefore solve a stronger soundness query (Definition 11) that pulls these quantifiers out to obtain the standard $\exists\forall$ formula with a quantifier-free body. The resulting formula can be solved with CEGIS [37], without requiring the underlying SMT solver to reason about quantifiers.

Definition 11 (Strongly Sound Mini Compiler). *Let* $\mathcal{A}_S = (\mathcal{I}_S, \Sigma_S, \mathcal{T}_S, \Phi_S)$ *and* $\mathcal{A}_T = (\mathcal{I}_T, \Sigma_T, \mathcal{T}_T, \Phi_T)$ *be two abstract register machines,* $\cong_{\mathcal{M}}$ *an injective state equivalence relation on their states* Σ_S *and* Σ_T*, and* $C : P(\iota) \rightarrow$

List$(P(\mathcal{I}_T))$ *a function for some* $\iota \in \mathcal{I}_S$. *We say that* C *is a* strongly sound mini compiler *for* $\iota_{\mathcal{M}}$ *with respect to* $\cong$ *iff*

$$\forall \sigma_S \in \Sigma_S,\ \sigma_T \in \Sigma_T,\ p \in P(\iota),\ a \in dom(mem(\sigma_S)),\ r \in dom(reg(\sigma_S)).$$
$$\sigma_S \cong_{\mathcal{M},a,r} \sigma_T \Rightarrow \mathcal{A}_S(p, \sigma_S) \cong_{\mathcal{M},a,r} \mathcal{A}_T(C(p), \sigma_T)$$

where $\cong_{\mathcal{M},a,r}$ *stands for the* $\cong_{\mathcal{M}}$ *formula with* a *and* r *as free variables.*

The JitSynth synthesis procedure is sound and complete with respect to this stronger query (Theorem 2). The proof follows from the soundness and completeness of the host synthesizer, and the construction of the compiler metasketch. We discharge this proof using Lean theorem prover [25].

Theorem 2 (Strong soundness and completeness of JitSynth). *Let* $\mathcal{C} = \text{CMS}(\iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M})$ *be the compiler metasketch for the abstract instruction* ι, *machines* $\mathcal{A}_S$ *and* $\mathcal{A}_T$, *and the state mapping* $\mathcal{M}$. *If* JitSynth *terminates and returns a program* C *when applied to* $\mathcal{C}$, *then* C *is a strongly sound mini compiler for* ι *and* $\mathcal{A}_T$ *(soundness). If there is a strongly sound mini compiler in the most general search space* $\{\text{Naive}(k, d, \iota, \mathcal{A}_S, \mathcal{A}_T, \mathcal{M}) \mid k, d \in \mathbb{N}\}$, *then* JitSynth *will terminate on* $\mathcal{C}$ *and produce a program (completeness).*

5 Implementation

We implemented JitSynth as described in Sect. 2 using Rosette [39] as our host language. Since the search spaces for different compiler lengths are disjoint, the JitSynth implementation searches these spaces in parallel [7]. We use $\Phi(\boldsymbol{p}) = \texttt{length}(\boldsymbol{p})$ as the fuel function for all languages studied in this paper. This provides sufficient fuel for evaluating programs in these languages that are accepted by the OS kernel. For example, the Linux kernel requires eBPF programs to be loop-free, and it enforces this restriction with a conservative static check; programs that fail the check are not passed to the JIT [13].

6 Evaluation

This section evaluates JitSynth by answering the following research questions:

RQ1: Can JitSynth synthesize correct and performant compilers for real-world source and target languages?
RQ2: How effective are the sketch optimizations described in Sect. 4?

6.1 Synthesizing Compilers for Real-World Source-Target Pairs

To demonstrate the effectiveness of JitSynth, we applied JitSynth to synthesize compilers for three different source-target pairs: eBPF to 64-bit RISC-V, classic BPF to eBPF, and libseccomp to eBPF. This subsection describes our results for each of the synthesized compilers.

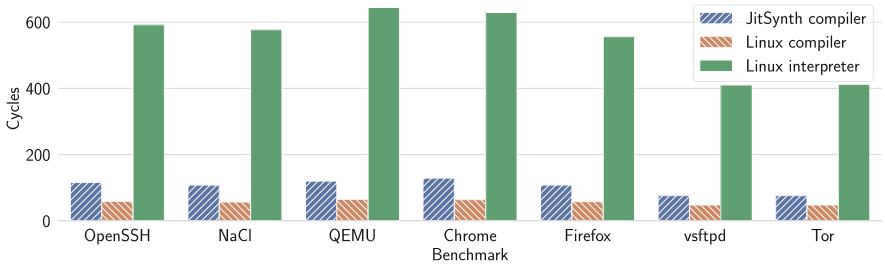


Fig. 7. Execution time of eBPF benchmarks on the HiFive Unleashed RISC-V development board, using the existing Linux eBPF to RISC-V compiler, the JitSynth compiler, and the Linux eBPF interpreter. Measured in processor cycles.

eBPF to RISC-V. As a case study, we applied JitSynth to synthesize a compiler from eBPF to 64-bit RISC-V. It supports 87 of the 102 eBPF instruction opcodes; unsupported eBPF instructions include function calls, endianness operations, and atomic instructions. To validate that the synthesized compiler is correct, we ran the existing eBPF test cases from the Linux kernel; our compiler passes all test cases it supports. In addition, our compiler avoids bugs previously found in the existing Linux eBPF-to-RISC-V compiler in Linux [27]. To evaluate performance, we compared against the existing Linux compiler. We used the same set of benchmarks used by Jitk [40], which includes system call filters from widely used applications. Because these benchmarks were originally for classic BPF, we first compile them to eBPF using the existing Linux classic-BPF-to-eBPF compiler as a preprocessing step. To run the benchmarks, we execute the generated code on the HiFive Unleashed RISC-V development board [35], measuring the number of cycles. As input to the filter, we use a system call number that is allowed by the filter to represent the common case execution.

Figure 7 shows the results of the performance evaluation. eBPF programs compiled by JitSynth JIT compilers show an average slowdown of 1.82× compared to programs compiled by the existing Linux compiler. This overhead results from additional complexity in the compiled eBPF jump instructions. Linux compilers avoid this complexity by leveraging bounds on the size of eBPF jump offsets. JitSynth-compiled programs get an average speedup of 5.24× compared to interpreting the eBPF programs. This evidence shows that JitSynth can synthesize a compiler that outperforms the current Linux eBPF interpreter, and nears the performance of the Linux compiler, while avoiding bugs.

Classic BPF to eBPF. Classic BPF is the original, simpler version of BPF used for packet filtering which was later extended to eBPF in Linux. Since many applications still use classic BPF, Linux must first compile classic BPF to eBPF as an intermediary step before compiling to machine instructions. As a second case study, we used JitSynth to synthesize a compiler from classic BPF to eBPF. Our synthesized compiler supports all classic BPF opcodes. To evaluate performance, we compare against the existing Linux classic-BPF-to-eBPF

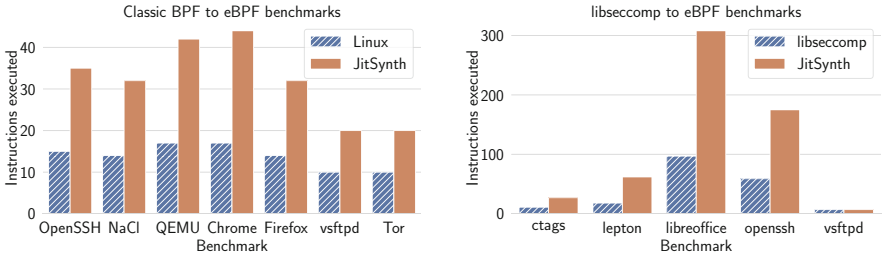


Fig. 8. Performance of code generated by JitSynth compilers compared to existing compilers for the classic BPF to eBPF benchmarks (left) and the libseccomp to eBPF benchmarks (right). Measured in number of instructions executed.

compiler. Similar to the RISC-V benchmarks, we run each eBPF program with input that is allowed by the filter. Because eBPF does not run directly on hardware, we measure the number of instructions executed instead of processor cycles.

Figure 8 shows the performance results. Classic BPF programs generated by JitSynth compilers execute an average of 2.28× more instructions than those compiled by Linux.

Libseccomp to eBPF. libseccomp is a library used to simplify construction of BPF system call filters. The existing libseccomp implementation compiles to classic BPF; we instead choose to compile to eBPF because classic BPF has only two registers, which does not satisfy the assumptions of JitSynth. Since libseccomp is a library and does not have distinct instructions, libseccomp itself does not meet the definition of an abstract register machine; we instead introduce an intermediate libseccomp language which does satisfy this definition. Our full libseccomp to eBPF compiler is composed of both a trusted program to translate from libseccomp to our intermediate language and a synthesized compiler from our intermediate language to eBPF.

To evaluate performance, we select a set of benchmark filters from real-world applications that use libseccomp, and measure the number of eBPF instructions executed for an input the filter allows. Because no existing compiler exists from libseccomp to eBPF directly, we compare against the composition of the existing libseccomp-to-classic-BPF and classic-BPF-to-eBPF compilers.

Figure 8 shows the performance results. libseccomp programs generated by JitSynth execute 2.61× more instructions on average compared to the existing libseccomp-to-eBPF compiler stack. However, the synthesized compiler avoids bugs previously found in the libseccomp-to-classic-BPF compiler [16].

6.2 Effectiveness of Sketch Optimizations

In order to evaluate the effectiveness of the search optimizations described in Sect. 4, we measured the time JitSynth takes to synthesize each of the three compilers with different optimizations enabled. Specifically, we run JitSynth in

Compiler	Naive sketch	RW sketch	PLD sketch
eBPF to RISC-V	X	X	44.4h
classic BPF to eBPF	X	X	1.2h
libseccomp to eBPF	4.0h	43.5m	7.1m

Fig. 9. Synthesis time for each source-target pair, broken down by set of optimizations used in the sketch. An X indicates that synthesis either timed out or ran out of memory.

three different configurations: (1) using Naive sketches, (2) using RW sketches, and (3) using PLD sketches. For each configuration, we ran JitSynth with a timeout of 48 hours (or until out of memory). Figure 9 shows the time to synthesize each compiler under each configuration. Note that these figures do not include time spent computing read and write sets, which takes less than 11 min for all cases. Our results were collected using an 8-core AMD Ryzen 7 1700 CPU with 16 GB memory, running Racket v7.4 and the Boolector [29] solver v3.0.1-pre.

When synthesizing the eBPF-to-RISC-V compiler, JitSynth runs out of memory with Naive sketches, reaches the timeout with RW sketches, and completes synthesis with PLD sketches. For the classic-BPF-to-eBPF compiler, JitSynth times out with both Naive sketches and RW sketches. JitSynth only finishes synthesis with PLD sketches. For the libseccomp-to-eBPF compiler, all configurations finish, but JitSynth finishes synthesis about 34× times faster with PLD sketches than with Naive sketches. These results demonstrate that the techniques JitSynth uses are essential to the scalability of JIT synthesis.

7 Related Work

JIT Compilers for In-kernel Languages. JIT compilers have been widely used to improve the extensibility and performance of systems software, such as OS kernels [8,11,12,26]. One notable system is Jitk [40]. It builds on the CompCert compiler [20] to compile classic BPF programs to machine instructions. Both Jitk and CompCert are formally verified for correctness using the Coq interactive theorem prover. Jitk is further extended to support eBPF [36]. Like Jitk, JitSynth provides formal correctness guarantees of JIT compilers. Unlike Jitk, JitSynth does not require developers to write either the implementation or proof of a JIT compiler. Instead, it takes as input interpreters of both source and target languages and state-mapping functions, using automated verification and synthesis to produce a JIT compiler.

An in-kernel extension system such as eBPF also contains a *verifier*, which checks for safety and termination of input programs [13,40]. JitSynth assumes a well-formed input program that passes the verifier and focuses on the correctness of JIT compilation.

Synthesis-Aided Compilers. There is a rich literature that explores generating and synthesizing peephole optimizers and superoptimizers based on a given ISA or language specification [4,9,14,17,23,33,34]. Bansal and Aiken described a PowerPC-to-x86 binary translator using peephole superoptimization [5]. Chlorophyll [31] applied synthesis to a number of compilation tasks for the GreenArrays GA144 architecture, including code partitioning, layout, and generation. JitSynth bears the similarity of translation between a source-target pair of languages and shares the challenge of scaling up synthesis. Unlike existing work, JitSynth synthesizes a *compiler* written in a host language, and uses compiler metasketches for efficient synthesis.

Compiler Testing. Compilers are complex pieces of software and are known to be difficult to get right [22]. Recent advances in compiler testing, such as Csmith [41] and EMI [42], have found hundreds of bugs in GCC and LLVM compilers. Alive [19,21] and Serval [28] use automated verification techniques to uncover bugs in the LLVM's peephole optimizer and the Linux kernel's eBPF JIT compilers, respectively. JitSynth complements these tools by providing a correctness-by-construction approach for writing JIT compilers.

8 Conclusion

This paper presents a new technique for synthesizing JIT compilers for in-kernel DSLs. The technique creates per-instruction compilers, or compilers that independently translate single source instructions to sequences of target instructions. In order to synthesize each per-instruction compiler, we frame the problem as search using compiler metasketches, which are optimized using both read and write set information as well as pre-synthesized load operations. We implement these techniques in JitSynth and evaluate JitSynth over three source and target pairs from the Linux kernel. Our evaluation shows that (1) JitSynth can synthesize correct and performant compilers for real in-kernel languages, and (2) the optimizations discussed in this paper make the synthesis of these compilers tractable to JitSynth. As future in-kernel DSLs are created, JitSynth can reduce both the programming and proof burden on developers writing compilers for those DSLs. The JitSynth source code is publicly available at https://github.com/uw-unsat/jitsynth.

References

1. Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2006
2. Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2011
3. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2014

4. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) [1], pp. 394–403 (2006)
5. Bansal, S., Aiken, A.: Binary translation using peephole superoptimizers. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, pp. 177–192, December 2008
6. Blazakis, D.: Interpreter exploitation: Pointer inference and JIT spraying. In: Black Hat DC, Arlington, VA, February 2010
7. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg, FL, pp. 775–788, January 2016
8. Chen, H., et al.: Security bugs in embedded interpreters. In: Proceedings of the 4th Asia-Pacific Workshop on Systems, 6 p. Singapore (2013)
9. Davidson, J.W., Fraser, C.W.: Automatic generation of peephole optimizations. In: Proceedings of the SIGPLAN Symposium on Compiler Construction, Montreal, Canada, pp. 111–116, June 1984
10. Edge, J.: A library for seccomp filters, April 2012. https://lwn.net/Articles/494252/
11. Engler, D.R.: VCODE: a retargetable, extensible, very fast dynamic code generation system. In: Proceedings of the 17th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadephia, PA, pp. 160–170, May 1996
12. Fleming, M.: A thorough introduction to eBPF, December 2017. https://lwn.net/Articles/740157/
13. Gershuni, E., et al.: Simple and precise static analysis of untrusted Linux kernel extensions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Phoenix, AZ , pp. 1069–1084, June 2019
14. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) [2], pp. 62–73 (2011)
15. Horn, J.: Issue 1454: arbitrary read+write via incorrect range tracking in eBPF, January 2018. https://bugs.chromium.org/p/project-zero/issues/detail?id=1454
16. Horn, J.: libseccomp: incorrect compilation of arithmetic comparisons, March 2019. https://bugs.chromium.org/p/project-zero/issues/detail?id=1769
17. Joshi, R., Nelson, G., Randall, K.: Denali: a goal-directed superoptimizer. In: Proceedings of the 23rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, pp. 304–314, June 2002
18. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: Proceedings of the 40th IEEE Symposium on Security and Privacy, San Francisco, CA, pp. 19–37, May 2019
19. Lee, J., Hur, C.K., Lopes, N.P.: AliveInLean: a verified LLVM peephole optimization verifier. In: Proceedings of the 31st International Conference on Computer Aided Verification (CAV), New York, NY, pp. 445–455, July 2019
20. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
21. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, OR, pp. 22–32, June 2015

22. Marcozzi, M., Tang, Q., Donaldson, A., Cadar, C.: Compiler fuzzing: how much does it matter? In: Proceedings of the 2019 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Athens, Greece, October 2019
23. Massalin, H.: Superoptimizer: a look at the smallest program. In: Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Palo Alto, CA, pp. 122–126, October 1987
24. McCanne, S., Jacobson, V.: The BSD packet filter: a new architecture for user-level packet capture. In: Proceedings of the Winter 1993 USENIX Technical Conference, San Diego, CA, pp. 259–270, January 1993
25. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 378–388. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26
26. Myreen, M.O.: Verified just-in-time compiler on x86. In: Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL), pp. 107–118. Association for Computing Machinery, New York, January 2010
27. Nelson, L.: bpf, riscv: clear high 32 bits for ALU32 add/sub/neg/lsh/rsh/arsh, May 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1e692f09e091
28. Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., Wang, X.: Scaling symbolic evaluation for automated verification of systems code with serval. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), Huntsville, Ontario, Canada, pp. 225–242, October 2019
29. Niemetz, A., Preiner, M., Biere, A.: Boolector 20 system description. J. Satisfiabil. Boolean Model. Comput. **9**, 53–58 (2014). (published 2015)
30. Paul, M.: CVE-2020-8835: linux kernel privilege escalation via improper eBPF program verification, April 2020. https://www.thezdi.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification
31. Phothilimthana, P.M., Jelvis, T., Shah, R., Totla, N., Chasins, S., Bodik, R.: Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) [3], pp. 396–407 (2014)
32. RISC-V Foundation: The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA, Document Version 2019121, December 2019
33. Sasnauskas, R., et al.: Souper: a synthesizing superoptimizer, November 2017. https://arxiv.org/abs/1711.04422
34. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Houston, TX, pp. 305–316, March 2013
35. SiFive: SiFive FU540-C000 manual, v1p0, April 2018. https://www.sifive.com/boards/hifive-unleashed
36. Sobel, L.: eJitk: extending Jitk to eBPF, May 2015. https://css.csail.mit.edu/6.888/2015/papers/ejitk_sobel.pdf
37. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) [1], pp. 404–415 (2006)

38. The Coq Development Team: The Coq Proof Assistant, version 8.9.0, January 2019. https://doi.org/10.5281/zenodo.2554024
39. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) [3], pp. 530–541 (2014)
40. Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., Tatlock, Z.: Jitk: a trustworthy in-kernel interpreter infrastructure. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, pp. 33–47, October 2014
41. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) [2], pp. 283–294 (2011)
42. Zhang, Q., Sun, C., Su, Z.: Skeletal program enumeration for rigorous compiler testing. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Barcelona, Spain, pp. 347–361 June 2017

Program Synthesis Using Deduction-Guided Reinforcement Learning

Yanju Chen[1(✉)], Chenglong Wang[2], Osbert Bastani[3], Isil Dillig[4], and Yu Feng[1(✉)]

[1] University of California, Santa Barbara, Santa Barbara, CA 93106, USA
{yanju,yufeng}@cs.ucsb.edu
[2] University of Washington, Seattle, WA 98115, USA
clwang@cs.washington.edu
[3] University of Pennsylvania, Philadelphia, PA 19104, USA
obastani@seas.upenn.edu
[4] The University of Texas at Austin, Austin, TX 78712, USA
isil@cs.utexas.edu

Abstract. In this paper, we present a new program synthesis algorithm based on reinforcement learning. Given an initial policy (i.e. statistical model) trained off-line, our method uses this policy to guide its search and gradually improves it by leveraging feedback obtained from a deductive reasoning engine. Specifically, we formulate program synthesis as a reinforcement learning problem and propose a new variant of the *policy gradient* algorithm that can incorporate feedback from a deduction engine into the underlying statistical model. The benefit of this approach is two-fold: First, it combines the power of deductive and statistical reasoning in a unified framework. Second, it leverages deduction not only to *prune* the search space but also to *guide* search. We have implemented the proposed approach in a tool called Concord and experimentally evaluate it on synthesis tasks studied in prior work. Our comparison against several baselines and two existing synthesis tools shows the advantages of our proposed approach. In particular, Concord solves 15% more benchmarks compared to Neo, a state-of-the-art synthesis tool, while improving synthesis time by 8.71× on benchmarks that can be solved by both tools.

1 Introduction

Due to its potential to significantly improve both programmer productivity and software correctness, *automated program synthesis* has gained enormous popularity over the last decade. Given a high-level specification of user intent, most modern synthesizers perform some form of backtracking search in order to find a

This work was sponsored by the National Science Foundation under agreement number of 1908494, 1811865 and 1910769.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 587–610, 2020.
https://doi.org/10.1007/978-3-030-53291-8_30

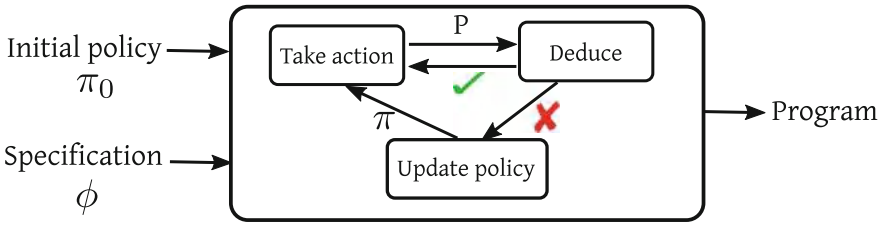


Fig. 1. Overview of our synthesis algorithm

program that satisfies the specification. However, due to the enormous size of the search space, synthesizers additionally use at least one of two other techniques, namely deduction and statistical reasoning, to make this approach practical. For example, many recent synthesis techniques use lightweight program analysis or logical reasoning to significantly prune the search space [18,19,39,53]. On the other hand, several recent approaches utilize a statistical model (trained off-line) to bias the search towards programs that are more likely to satisfy the specification [2,4,7,19]. While both deductive and statistical reasoning have been shown to dramatically improve synthesis efficiency, a key limitation of existing approaches is that they do not tightly combine these two modes of reasoning. In particular, although logical reasoning often provides very useful feedback at synthesis time, existing synthesis algorithms do not leverage such feedback to improve their statistical model.

In this paper, we propose a new synthesis algorithm that meaningfully combines deductive and statistical reasoning. Similar to prior techniques, our approach starts with a statistical model (henceforth called a *policy*) that is trained off-line on a representative set of training problems and uses this policy to guide search. However, unlike prior techniques, our method *updates* this policy on-line at synthesis time and gradually improves the policy by incorporating feedback from a deduction engine.

To achieve this tight coupling between deductive and statistical reasoning, we formulate syntax-guided synthesis as a reinforcement learning (RL) problem. Specifically, given a context-free grammar for the underlying DSL, we think of partial (i.e., incomplete) programs in this DSL as states in a Markov Decision Process (MDP) and actions as grammar productions. Thus, a *policy* of this MDP specifies how a partial program should be extended to obtain a more specific program. Then, the goal of our reinforcement learning problem is to improve this policy over time as some partial programs are proven infeasible by an underlying deduction engine.

While the framework of reinforcement learning is a good fit for our problem, standard RL algorithms (e.g., policy gradient) typically update the policy based on feedback received from states that have *already* been explored. However, in the context of program synthesis, deductive reasoning can also provide feedback about states that have *not* been explored. For example, given a partial program that is infeasible, one can analyze the root cause of failure to infer *other* infeasible

programs [18,54]. To deal with this difficulty, we propose an *off-policy* reinforcement learning algorithm that can improve the policy based on such additional feedback from the deduction engine.

As shown schematically in Fig. 1, our synthesis algorithm consists of three conceptual elements, indicated as "Take action", "Deduce", and "Update policy". Given the current policy π and partial program P, "Take action" uses π to expand P into a more complete program P'. Then, "Deduce" employs existing deductive reasoning techniques (e.g., [18,32]) to check whether P' is feasible with respect to the specification. If this is not the case, "Update policy" uses the feedback provided by the deduction engine to improve π. Specifically, the policy is updated using an off-policy variant of the *policy gradient* algorithm, where the gradient computation is adapted to our unique setting.

We have implemented the proposed method in a new synthesis tool called Concord and empirically evaluate it on synthesis tasks used in prior work [2,18]. We also compare our method with several relevant baselines as well as two existing synthesis tools. Notably, our evaluation shows that Concord can solve 15% more benchmarks compared to Neo (a state-of-the-art synthesis tool), while being 8.71× faster on benchmarks that can be solved by both tools. Furthermore, our ablation study demonstrates the empirical benefits of our proposed reinforcement learning algorithm.

To summarize, this paper makes the following key contributions:

- We propose a new synthesis algorithm based on reinforcement learning that tightly couples statistical and deductive reasoning.
- We describe an off-policy reinforcement learning technique that uses the output of the deduction engine to gradually improve its policy.
- We implement our approach in a tool called Concord and empirically demonstrate its benefits compared to other state-of-the-art tools as well as ablations of our own system.

The rest of this paper is structured as follows. First, we provide some background on reinforcement learning and MDPs (Sect. 2) and introduce our problem formulation in Sect. 3. After formulating the synthesis problem as an MDP in Sect. 4, we then present our synthesis algorithm in Sect. 5. Sections 6 and 7 describe our implementation and evaluation respectively. Finally, we discuss related work and future research directions in Sect. 8 and 9.

2 Background on Reinforcement Learning

At a high level, the goal of reinforcement learning (RL) is to train an agent, such as a robot, to make a sequence of decisions (e.g., move up/down/left/right) in order to accomplish a task. All relevant information about the environment and the task is specified as a *Markov decision process (MDP)*. Given an MDP, the goal is to compute a policy that specifies how the agent should act in each state to maximize their chances of accomplishing the task.

In the remainder of this section, we provide background on MDPs and describe the policy gradient algorithm that our method will build upon.

Markov Decision Process. We formalize a *Markov decision process (MDP)* as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{S}_I, \mathcal{S}_T, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where:

- $\mathcal{S}$ is a set of *states* (e.g., the robot's current position),
- $\mathcal{S}_I$ is the initial state distribution,
- $\mathcal{S}_T$ is a set of the final states (e.g., a dead end),
- $\mathcal{A}$ is a set of actions (e.g., move up/down/left/right),
- $\mathcal{F} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a set of transitions,
- $\mathcal{R} : \mathcal{S} \to \mathbb{R}$ is a reward function that assigns a reward to each state (e.g., 1 for reaching the goal and 0 otherwise).

In general, transitions in an MDP can be stochastic; however, for our setting, we only consider deterministic transitions and rewards.

Policy. A policy for an MDP specifies how the agent should act in each state. Specifically, we consider a (stochastic) *policy* $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where $\pi(S, A)$ is the probability of taking action A in state S. Alternatively, we can also think of π as a mapping from states to distributions over actions. Thus, we write $A \sim \pi(S)$ to denote that action A is sampled from the distribution for state s.

Rollout. Given an MDP $\mathcal{M}$ and policy π, a *rollout* is a sequence of state-action-reward tuples obtained by sampling an initial state and then using π to make decisions until a final state is reached. More formally, for a rollout of the form:

$$\zeta = ((S_1, A_1, R_1), ..., (S_{m-1}, A_{m-1}, R_{m-1}), (S_m, \varnothing, R_m)),$$

we have $S_m \in \mathcal{S}_T$, $S_1 \sim \mathcal{S}_I$ (i.e., S_1 is sampled from an initial state), and, for each $i \in \{1, ..., m-1\}$, $A_i \sim \pi(S_i)$, $R_i = \mathcal{R}(S_i)$, and $S_{i+1} = \mathcal{F}(S_i, A_i)$.

In general, a policy π induces a distribution $\mathcal{D}_\pi$ over the rollouts of an MDP $\mathcal{M}$. Since we assume that MDP transitions are deterministic, we have:

$$\mathcal{D}_\pi(\zeta) = \prod_{i=1}^{m-1} \pi(S_i, A_i).$$

RL Problem. Given an MDP $\mathcal{M}$, the goal of reinforcement learning is to compute an *optimal* policy π^* for $\mathcal{M}$. More formally, π^* should maximize *cumulative expected reward*:

$$\pi^* = \arg\max_\pi J(\pi)$$

where the *cumulative expected reward* $J(\pi)$ is computed as follows:

$$J(\pi) = \mathbb{E}_{\zeta \sim \mathcal{D}_\pi} \left[\sum_{i=1}^{m} R_i \right]$$

Policy Gradient Algorithm. The *policy gradient algorithm* is a well-known RL algorithm for finding optimal policies. It assumes a parametric policy family π_θ with parameters $\theta \in \mathbb{R}^d$. For example, π_θ may be a deep neural network (DNN), where θ denotes the parameters of the DNN. At a high level, the policy gradient algorithm uses the following theorem to optimize $J(\pi_\theta)$ [48]:

Theorem 1. *We have*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\zeta \sim \mathcal{D}_{\pi_\theta}}[\ell(\zeta)] \qquad \textit{where} \qquad \ell(\zeta) = \sum_{i=1}^{m-1} \left(\sum_{j=i+1}^{m} R_j \right) \nabla_\theta \log \pi_\theta(S_i, A_i). \tag{1}$$

In this theorem, the term $\nabla_\theta \log \pi_\theta(S_i, A_i)$ intuitively gives a direction in the parameter space that, when moving the policy parameters towards it, increases the probability of taking action A_i at state S_i. Also, the sum $\sum_{j=i+1}^{m} R_j$ is the total future reward after taking action A_i. Thus, $\ell(\zeta)$ is just the sum of different directions in the parameter space weighted by their corresponding future reward. Thus, the gradient $\nabla_\theta J(\pi_\theta)$ moves policy parameters in a direction that increases the probability of taking actions that lead to higher rewards.

Based on this theorem, we can estimate the gradient $\nabla_\theta J(\pi_\theta)$ using rollouts sampled from $\mathcal{D}_{\pi_\theta}$:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{n} \sum_{k=1}^{n} \ell(\zeta^{(k)}), \tag{2}$$

where $\zeta^{(k)} \sim \mathcal{D}_{\pi_\theta}$ for each $k \in \{1, ..., n\}$. The policy gradient algorithm uses stochastic gradient ascent in conjunction with Eq. (2) to maximize $J(\pi_\theta)$ [48].

3 Problem Formulation

In this paper, we focus on the setting of syntax-guided synthesis [1]. Specifically, given a domain-specific language (DSL) L and a specification ϕ, our goal is to find a program in L that satisfies ϕ. In the remainder of this section, we formally define our synthesis problem and clarify our assumptions.

DSL. We assume a domain-specific language L specified as a context-free grammar $L = (V, \Sigma, R, S)$, where V, Σ denote non-terminals and terminals respectively, R is a set of productions, and S is the start symbol.

Definition 1 (Partial program). *A* partial program P *is a sequence* $P \in (\Sigma \cup V)^*$ *such that* $S \overset{*}{\Rightarrow} P$ *(i.e., P can be derived from S via a sequence of productions). We refer to any non-terminal in P as a hole* hole*, and we say that P is* complete *if it does not contain any holes.*

$$
\begin{aligned}
S &\rightarrow N \mid L \\
N &\rightarrow 0 \mid \ldots \mid 10 \mid x_i \\
L &\rightarrow x_i \mid \texttt{take}(L, N) \mid \texttt{drop}(L, N) \mid \texttt{sort}(L) \\
&\quad \mid \texttt{reverse}(L) \mid \texttt{add}(L, L) \mid \texttt{sub}(L, L) \mid \texttt{sumUpTo}(L)
\end{aligned}
$$

Fig. 2. A simple programming language used for illustration. Here, `take` (resp. `drop`) keeps (resp. removes) the first N elements in the input list. Also, `add` (resp. `sub`) compute a new list by adding (resp. subtracting) elements from the two lists pair-wise. Finally, `sumUpTo` generates a new list where the i'th element in the output list is the sum of all previous elements (including the i'th element) in the input list.

Given a partial program P containing a hole H, we can fill this hole by replacing H with the right-hand-side of any grammar production r of the form $H \rightarrow e$. We use the notation $P \overset{r}{\Rightarrow} P'$ to indicate that P' is the partial program obtained by replacing the first occurrence of H with the right-hand-side of r, and we write $\textsc{Fill}(P, r) = P'$ whenever $P \overset{r}{\Rightarrow} P'$.

Example 1. Consider the small programming language shown in Fig. 2 for manipulating lists of integers. The following partial program P over this DSL contains three holes, namely L_1, L_2, N_1:

$$\texttt{add}(L_1, \texttt{take}(L_2, N_1))$$

Now, consider the production $r \equiv L \rightarrow \texttt{reverse}(L)$. In this case, $\textsc{Fill}(P, r)$ yields the following partial program P':

$$\texttt{add}(\texttt{reverse}(L_1), \texttt{take}(L_2, N_1))$$

Program Synthesis Problem. Given a specification ϕ and language $L = (V, \Sigma, R, S)$, the goal of program synthesis is to find a *complete* program P such that $S \overset{*}{\Rightarrow} P$ and P satisfies ϕ. We use the notation $P \models \phi$ to indicate that P is a complete program that satisfies specification ϕ.

Deduction Engine. In the remainder of this paper, we assume access to a *deduction engine* that can determine whether a partial program P is *feasible* with respect to specification ϕ. To make this more precise, we introduce the following notion of feasibility.

Definition 2 (Feasible partial program). *Given a specification ϕ and language $L = (V, \Sigma, R, S)$, a partial program P is said to be* feasible *with respect to ϕ if there exists any complete program P' such that $P \overset{*}{\Rightarrow} P'$ and $P' \models \phi$.*

In other words, a feasible partial program can be refined into a complete program that satisfies the specification. We assume that our deduction oracle over-approximates feasibility. That is, if P is feasible with respect to specification ϕ, then $\textsc{Deduce}(P, \phi)$ should report that P is feasible but not necessarily vice versa. Note that almost all deduction techniques used in the program synthesis literature satisfy this assumption [18,19,21,27,53].

Example 2. Consider again the DSL from Fig. 2 and the specification ϕ defined by the following input-output example:

$$[65, 2, 73, 62, 78] \mapsto [143, 129, 213, 204, 345]$$

The partial program $\texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N))$ is infeasible because, no matter what production we use to fill non-terminal N, the resulting program cannot satisfy the provided specification for the following reason:

- Given a list l and integer n where $n < \mathit{length}(l)$, $\texttt{take}(l, n)$ returns the first n elements in l. Thus, the length of $\texttt{take}(l, n)$ is smaller than that of l.
- The construct $\texttt{reverse}(l)$ reverses its input; thus, the size of the output list is the same as its input.
- Finally, $\texttt{add}(l_1, l_2)$ constructs a new list by adding the elements of its input lists pair-wise. Thus, `add` expects the two input lists to be the same size.
- Since the outputs of `reverse` and `take` do not have the same size, we cannot combine them using `add`.

Several techniques from prior work (e.g., [18,19,39,53]) can prove the infeasibility of such partial programs by using an SMT solver (provided specifications are given for the DSL constructs).

Beyond checking feasibility, some deduction techniques used for synthesis can also provide additional information [18,32,54]. In particular, given a partial program P that is infeasible with respect to specification ϕ, several deduction engines can generate a set of other infeasible partial programs $P_1, \ldots, P_n$ that are infeasible for the same reason as P. To unify both types of feedback, we assume that the output of the deduction oracle $\mathcal{O}$ is a set S of partial programs such that S is empty if and only if $\mathcal{O}$ decides that the partial program is feasible.

This discussion is summarized by the following definition:

Definition 3 (Deduction engine). *Given a partial program P and specification ϕ,* Deduce*(P, ϕ) yields a set of partial programs S such that (1) if $S \neq \varnothing$, then P is infeasible, and (2) for every $P' \in S$, it must be the case that P' is infeasible with respect to ϕ.*

Example 3. Consider again the same infeasible partial program P given in Example 2. Since $\texttt{drop}(l, n)$ drops the first n elements from list l (where $n < \mathit{length}(l)$), it also produces a list whose length is smaller than that of the input. Thus, the following partial program P' is also infeasible for the same reason as P:

$$P' \equiv \texttt{add}(\texttt{reverse}(x), \texttt{drop}(x, N))$$

Thus, Deduce(P, ϕ) may return the set $\{P, P'\}$.

4 MDP Formulation of Deduction-Guided Synthesis

Given a specification ϕ and language $L = (V, \Sigma, R, S)$, we can formulate the program synthesis problem as an MDP $\mathcal{M}_\phi = (\mathcal{S}, \mathcal{S}_I, \mathcal{S}_T, \mathcal{A}, \mathcal{F}, \mathcal{R})$, where:

- States $\mathcal{S}$ include all partial programs P such that $S \overset{*}{\Rightarrow} P$ as well as a special label $\perp$ indicating a syntactically ill-formed partial program
- $\mathcal{S}_I$ places all probability mass on the empty program S, i.e.,

$$\mathcal{S}_I(P) = \begin{cases} 1 \text{ if } P = S \\ 0 \text{ if } P \neq S \end{cases}$$

- $\mathcal{S}_T$ includes complete programs as well as infeasible partial programs, i.e.,

$$P \in \mathcal{S}_T \iff \textsc{IsComplete}(P) \ \vee \ \textsc{Deduce}(P, \phi) \neq \varnothing \vee P = \perp$$

- Actions $\mathcal{A}$ are exactly the productions R for the DSL
- Transitions $\mathcal{F}$ correspond to filling a hole using some production i.e.,

$$\mathcal{F}(P,\ r = (H \to e)) = \begin{cases} \perp & \text{if } H \text{ is not a hole in } P \\ \textsc{Fill}(P, r) & \text{otherwise} \end{cases}$$

- The reward function penalizes infeasible programs and rewards correct solutions, i.e.,

$$\mathcal{R}(P) = \begin{cases} 1 & \text{if } P \models \phi \\ -1 & \text{if } P = \perp \vee \textsc{Deduce}(P, \phi) \neq \varnothing \vee (\textsc{IsComplete}(P) \wedge P \not\models \phi) \\ 0 & \text{otherwise.} \end{cases}$$

Observe that our reward function encodes the goal of synthesizing a complete program P that satisfies ϕ, while avoiding the exploration of as many infeasible programs as possible. Thus, if we have a good policy π for this MDP, then a rollout of π is likely to correspond to a solution of the given synthesis problem.

Example 4. Consider the same specification (i.e., input-output example) ϕ from Example 2 and the DSL from Example 1. The partial program

$$P \equiv \texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N))$$

is a terminal state of $\mathcal{M}_\phi$ since $\textsc{Deduce}(P, \phi)$ yields a non-empty set, and we have $\mathcal{R}(P) = -1$. Thus, the following sequence corresponds to a rollout of $\mathcal{M}_\phi$:

$$(S, S \to L, 0),\ (L, L \to \texttt{add}(L, L), 0),\ (\texttt{add}(L_1, L_2), L \to \texttt{reverse}(L), 0)$$
$$(\texttt{add}(\texttt{reverse}(L_1), L_2), L \to x, 0),\ (\texttt{add}(\texttt{reverse}(x), L), L \to \texttt{take}(L, N), 0)$$
$$(\texttt{add}(\texttt{reverse}(x), \texttt{take}(L, N)), L \to x, 0), (\texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N)), \varnothing, -1).$$

Simplified Policy Gradient Estimate for $\mathcal{M}_\phi$. Since our synthesis algorithm will be based on policy gradient, we will now derive a simplified policy gradient for our MDP $\mathcal{M}_\phi$. First, by construction of $\mathcal{M}_\phi$, a rollout ζ has the form

$$(P_1, r_1, 0), ..., (P_m, \varnothing, q)$$

where $q = 1$ if $P_m \models \phi$ and $q = -1$ otherwise. Thus, the term $\ell(P)$ from Eq. 1 can be simplified as follows:

$$\ell(P_m) = \sum_{i=1}^{m-1} q \cdot \nabla_\theta \log \pi_\theta(P_i, r_i), \tag{3}$$

where $P_m \sim \mathcal{D}_{\pi_\theta}$ is a final state (i.e., complete program or infeasible partial program) sampled using π_θ. Then, Eq. 1 is equivalently

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{n} \sum_{k=1}^{n} \ell(P^{(k)}), \tag{4}$$

where $P^{(k)} \sim \mathcal{D}_{\pi_\theta}$ for each $k \in \{1, ..., n\}$.

5 RL-Based Synthesis Algorithm

In this section, we describe our synthesis algorithm based on reinforcement learning. Our method is an *off-policy* variant of the standard (on-policy) policy gradient algorithm and incorporates additional feedback – in the form of other infeasible programs – provided by the deduction engine when improving its policy parameters. We first give a high-level overview of the synthesis algorithm and then explain how to update the policy.

5.1 Overview of Synthesis Algorithm

Our RL-based synthesis algorithm is presented in Fig. 3. In addition to specification ϕ and domain-specific language L, this algorithm also takes as input an initial policy π_0 that has been trained off-line on a representative set of training problems.[1] In each iteration of the main synthesis loop, we first obtain a rollout of the current policy by calling the GetRollout procedure at line 7. Here, each rollout either corresponds to a complete program P or an infeasible partial program. If P is complete *and* satisfies the specification, we return it as a solution in line 8. Otherwise, we use feedback $\mathcal{C}$ provided by the deduction engine to improve the current policy (line 9). In the following subsections, we explain the GetRollout and UpdatePolicy procedures in more detail.

5.2 Sampling Rollouts

The GetRollout procedure iteratively expands a partial program, starting from the start symbol S of the grammar (line 11). In each iteration (lines 12–19), we first check whether the current partial program P is feasible by calling Deduce. If P is infeasible (i.e., $\mathcal{C}$ is non-empty), then we have reached a terminal

[1] We explain how to train this initial policy in Sect. 6.

```
1:  procedure SYNTHESIZE(L, φ, π_0)
2:      input: Domain-specific language L = (V, Σ, R, S)
3:      input: Specification φ; initial policy π_0
4:      output: Complete program P such that P ⊨ φ
5:      π_θ ← π_0
6:      while true do
7:          (P, C) ← GETROLLOUT(L, φ, π_θ)
8:          if C = ∅ then return P
9:          else π_θ ← UPDATEPOLICY(π_θ, C)
10: procedure GETROLLOUT(L, φ, π_θ)
11:     P ← S
12:     while true do
13:         C ← DEDUCE(P, φ)
14:         if C ≠ ∅ then return (P, C)
15:         choose r ∼ π_θ(P) ∧ LHS(r) ∈ HOLES(P)
16:         P ← FILL(P, r)
17:         if ISCOMPLETE(P) then
18:             if P ⊨ φ then return (P, ∅)
19:             else return (P, {P})
20: procedure UPDATEPOLICY(π_θ, C)
21:     for k ∈ {1, ..., n'} do
22:         P^(k) ∼ Uniform(C)
23:     θ' ← θ + η Σ_{k=1}^{n'} ℓ(P^(k)) · D_{π_θ}(P^(k)) / (1/|C|)
24:     return π_θ'
```

Fig. 3. Deduction-guided synthesis algorithm based on reinforcement learning

state of the MDP; thus, we return P as the final state of the rollout. Otherwise, we continue expanding P according to the current policy π_θ. Specifically, we first sample an action (i.e., grammar production) r that is applicable to the current state (i.e., the left-hand-side of r is a hole in P), and, then, we expand P by calling the FILL procedure (defined in Sect. 3) at line 16. If the resulting program is complete, we have reached a terminal state and return P; otherwise, we continue expanding P according to the current policy.

5.3 Improving the Policy

As mentioned earlier, our algorithm improves the policy by using the feedback $\mathcal{C}$ provided by the deduction engine. Specifically, consider an infeasible program P explored by the synthesis algorithm at line 7. Since DEDUCE(P, ϕ) yields a set of infeasible programs, for every program $P' \in \mathcal{C}$, we know that the reward should be -1. As a consequence, we should be able to incorporate the rollout used to construct P into the policy gradient estimate based on Eq. (3). However, the challenge to doing so is that Eq. (4) relies on *on-policy* samples – i.e., the

programs $P^{(k)}$ in Eq. (4) must be sampled using the current policy π_θ. Since $P' \in \mathcal{C}$ is not sampled using π_θ, we cannot directly use it in Eq. (4).

Instead, we use *off-policy* RL to incorporate P' into the estimate of $\nabla_\theta J(\pi_\theta)$ [28]. Essentially, the idea is to use *importance weighting* to incorporate data sampled from a different distribution than $\mathcal{D}_{\pi_\theta}$. In particular, suppose we are given a distribution $\tilde{\mathcal{D}}$ over final states. Then, we can derive the following gradient:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{P\sim\mathcal{D}_{\pi_\theta}}[\ell(P)] \\ &= \mathbb{E}_{P\sim\tilde{\mathcal{D}}}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{\tilde{\mathcal{D}}(P)}\right]\end{aligned} \tag{5}$$

Intuitively, the *importance weight* $\frac{\mathcal{D}_{\pi_\theta}(P)}{\tilde{\mathcal{D}}(P)}$ accounts for the fact that P is sampled from the "wrong" distribution.

Now, we can use the distribution $\tilde{\mathcal{D}} = \text{Uniform}(\textsc{Deduce}(P', \phi))$ for a randomly sampled final state $P' \sim \mathcal{D}_{\pi_\theta}$. Thus, we have[2]:

Theorem 2. *The policy gradient is*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}, P\sim \mathit{Uniform}(\textsc{Deduce}(P',\phi))}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{1/|\textsc{Deduce}(P',\phi)|}\right]. \tag{6}$$

Proof. Note that

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}}[\nabla_\theta J(\pi_\theta)] \\ &= \mathbb{E}_{P'\sim\mathcal{D}_{\pi_\theta}, P\sim\text{Uniform}(\textsc{Deduce}(P',\phi))}\left[\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P)}{1/|\textsc{Deduce}(P',\phi)|}\right],\end{aligned}$$

as claimed. □

The corresponding estimate of $\nabla_\theta J(\pi_\theta)$ is given by the following equation:

$$\nabla_\theta J(\theta) \approx \frac{1}{n}\sum_{k=1}^{n}\frac{1}{n'}\sum_{k'=1}^{n'}\ell(P^{(k,k')})\cdot\frac{\mathcal{D}_{\pi_\theta}(P^{(k,k')})}{1/|\textsc{Deduce}(P^{(k)},\phi)|},$$

where $P^{(k)} \sim \tilde{\mathcal{D}}$ and $P^{(k,k')} \sim \text{Uniform}(\textsc{Deduce}(P^{(k)}, \phi))$ for each $k \in \{1, ..., n\}$ and $k' \in \{1, ..., n'\}$. Our actual implementation uses $n = 1$, in which case this equation can be simplified to the following:

$$\nabla_\theta J(\theta) \approx \frac{1}{n'}\sum_{k'=1}^{n'}\ell(P)\cdot\frac{\mathcal{D}_{\pi_\theta}(P^{(k')})}{1/|\textsc{Deduce}(P,\phi)|}, \tag{7}$$

[2] Technically, importance weighting requires that the support of $\tilde{\mathcal{D}}$ contains the support of $\mathcal{D}_{\pi_\theta}$. We can address this issue by combining $\tilde{\mathcal{D}}$ and $\mathcal{D}_{\pi_\theta}$—in particular, take $\tilde{\mathcal{D}}(P) = (1-\epsilon)\cdot\text{Uniform}(\textsc{Deduce}(P',\phi))(P) + \epsilon\cdot\mathcal{D}_{\pi_\theta}(P)$, for any $\epsilon > 0$.

where $P \sim \tilde{\mathcal{D}}$ and $P^{(k')} \sim \text{Uniform}(\textsc{Deduce}(P, \phi))$ for each $k' \in \{1, ..., n'\}$.

Now, going back to our synthesis algorithm from Fig. 3, the UpdatePolicy procedure uses Eq. 7 to update the policy parameters θ. Specifically, given a set $\mathcal{C}$ of infeasible partial programs, we first sample n' programs $P^{(1)}, \ldots, P^{(n')}$ from $\mathcal{C}$ uniformly at random (line 22). Then, we use the probability of each $P^{(k)}$ being sampled from the current distribution $\mathcal{D}_{\pi_\theta}$ to update the policy parameters to a new value θ' according to Eq. 7.

Example 5. Suppose that the current policy assigns the following probabilities to these state, action pairs:

$$\pi_\theta((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{take}(L, N)) = 0.3$$
$$\pi_\theta((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{drop}(L, N)) = 0.3$$
$$\pi_\theta((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{sumUpTo}(L)) = 0.1$$

Furthermore, suppose that we sample the following rollout using this policy:

$$P \equiv \texttt{add}(\texttt{reverse}(x), \texttt{take}(x, N)),$$

This corresponds to an infeasible partial program, and, as in Example 3, $\textsc{Deduce}(P, \phi)$ yields $\{P, P'\}$ where $P' \equiv \texttt{add}(\texttt{reverse}(x), \texttt{drop}(x, N))$. Using the gradients derived by Eq. 7, we update the policy parameters θ to θ'. The updated policy now assigns the following probabilities to the same state, action pairs:

$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{take}(L, N)) = 0.15$$
$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{drop}(L, N)) = 0.15$$
$$\pi_{\theta'}((\texttt{add}(\texttt{reverse}(x), L)), L \rightarrow \texttt{sumUpTo}(L)) = 0.2$$

Observe that the updated policy makes it less likely that we will expand the partial program $\texttt{add}(\texttt{reverse}(x), L))$ using the `drop` production in addition to the `take` production. Thus, if we reach the same state $\texttt{add}(\texttt{reverse}(x), L)$ during rollout sampling in the next iteration, the policy will make it more likely to explore the `sumUpTo` production, which does occur in the desired program

$$\texttt{add}(\texttt{reverse}(x), \texttt{sumUpTo}(x))$$

that meets the specification from Example 2.

6 Implementation

We have implemented the proposed algorithm in a new tool called Concord written in Python. In what follows, we elaborate on various aspects of our implementation.

6.1 Deduction Engine

CONCORD uses the same deduction engine described by Feng et al. [18]. Specifically, given a partial program P, CONCORD first generates a specification φ of P by leveraging the abstract semantics of each DSL construct. Then, CONCORD issues a satisfiability query to the Z3 SMT solver [15] to check whether φ is consistent with the provided specification. If it is not, this means that P is infeasible, and CONCORD proceeds to infer other partial programs that are also infeasible for the same reason as P. To do so, CONCORD first obtains an unsatisfiable core ψ for the queried formula, and, for each clause c_i of ψ originating from DSL construct f_i, it identifies a set S_i of other DSL constructs whose semantics imply c_i. Finally, it generates a set of other infeasible programs by replacing all f_i's in the current program with another construct drawn from its corresponding set S_i.

6.2 Policy Network

Architecture. As shown by Fig. 4, CONCORD represents its underlying policy using a deep neural network (DNN) $\pi_\theta(r \mid P)$, which takes as input the current state (i.e., a partial program P) and outputs a probability distribution over actions (i.e., productions r in the DSL). We represent each program P as a flat sequence of statements and use a recurrent neural network (RNN) architecture, as this is a natural choice for sequence inputs. In particular, our policy network is a gated recurrent unit (GRU) network [13], which is a state-of-the-art RNN architecture. Our policy network has one hidden layer with 256 neurons; this layer is sequentially applied to each statement in the partial program together with the latent vector from processing the previous statement. Once the entire partial program P has been encoded into a vector, π_θ has a final layer that outputs a distribution over DSL productions r based on this vector.

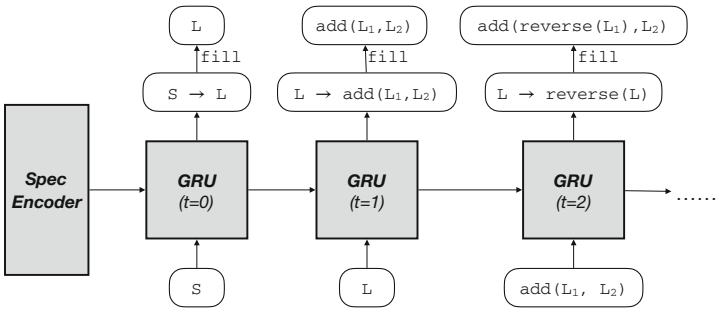


Fig. 4. The architecture of the policy network showing how to roll out the partial program in Example 4.

Pretraining the Initial Policy. Recall from Sect. 5 that our synthesis algorithm takes a input an *initial policy network* that is updated during the synthesis

process. One way to initialize the the policy network would be to use a standard random initialization of the network weights. However, a more effective alternative is to *pretrain* the policy on a benchmark suite of program synthesis problems [44]. Specifically, consider a representative training set X_{train} of synthesis problems of the form (ϕ, P), where ϕ is the specification and P is the desired program. To obtain an initial policy, we augment our policy network to take as input an encoding of the specification ϕ for the current synthesis problem – i.e., it has the form $\pi_\theta(r \mid P, \phi)$.[3] Then, we use supervised learning to train π_θ to predict P given ϕ—i.e.,

$$\theta^0 = \arg\max_\theta \sum_{(\phi,P)\in X_{\text{train}}} \sum_{i=1}^{|P|-1} \pi_\theta(r_i \mid P_i, \phi).$$

We optimize θ using stochastic-gradient descent (SGD) on this objective.

Given a new synthesis problem ϕ, we use π_{θ^0} as the initial policy. Our RL algorithm then continues to update the parameters starting from θ^0.

6.3 Input Featurization

As standard, we need a way to featurize the inputs to our policy network – i.e., the statements in each partial program P, and the specification ϕ. Our current implementation assumes that statements are drawn from a finite set and featurizes them by training a different embedding vector for each kind of statement. While our general methodology can be applied to different types specifications, our implementation featurizes the specification under the assumption that it consists of input-output examples and uses the same methodology described by Balog et al. [2].

6.4 Optimizations

Our implementation performs a few optimization over the algorithm presented in Sect. 5. First, since it is possible to sample the same rollout multiple times, our implementation uses a hash map to check whether a rollout has already been explored. Second, in different invocations of the GetRollout procedure from Fig. 3, we may end up querying the feasibility of the same state (i.e., partial program) *many* times. Since checking feasibility requires a potentially-expensive call to the SMT solver, our implementation also memoizes the results of feasibility checks for each state. Finally, similar to Chen et al. [11], we use a 3-model ensemble to alleviate some of the randomness in the synthesis process and return a solution as soon as one of the models in the ensemble finds a correct solution.

7 Evaluation

In this section, we describe the results from our experimental evaluation, which is designed to answer the following key research questions:

[3] Including the specification as an input to π_θ is unnecessary if we do not use pretraining, since ϕ does not change for a single synthesis problem.

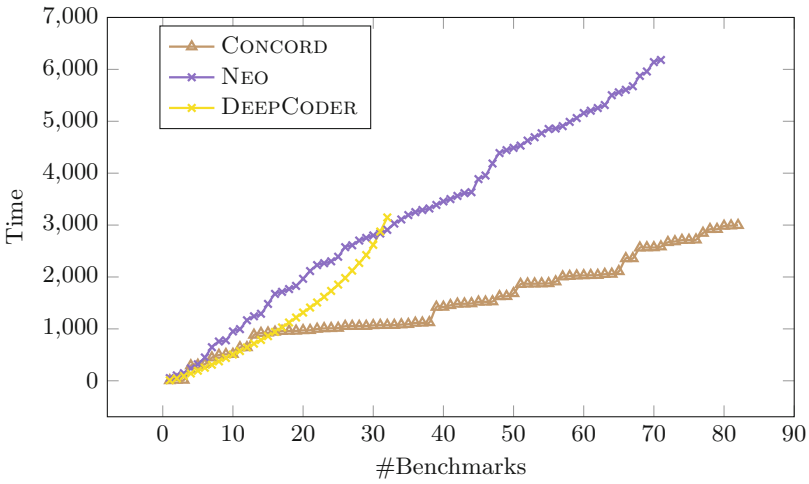


Fig. 5. Comparison between Concord, Neo, and DeepCoder

1. How does Concord compare against existing synthesis tools?
2. What is the impact of updating the statistical model during synthesis? (i.e., is reinforcement learning actually useful)?
3. How important is the proposed off-policy RL algorithm compared to standard policy gradient?
4. How important is it to get feedback from the deduction engine when updating the policy?

Benchmarks. We evaluate the proposed technique on a total of 100 synthesis tasks used in prior work [2,18]. Specifically, these synthesis tasks require performing non-trivial transformations and computations over lists using a functional programming language. Since these benchmarks have been used to evaluate both Neo [18] and DeepCoder [2], they provide a fair ground for comparing our approach against two of the most closely-related techniques. In particular, note that DeepCoder uses a pre-trained deep neural network to guide its search, whereas Neo uses both statistical and logical reasoning (i.e., statistical model to guide search and deduction to prune the search space). However, unlike our proposed approach, neither Neo nor DeepCoder update their statistical model during synthesis time.

Training. Recall that our algorithm utilizes a pre-trained initial policy. To generate the initial policy, we use the same methodology described in DeepCoder [2] and adopted in Neo [18]. Specifically, we randomly generate both programs and inputs, and we obtain the corresponding output by executing the program. Then, we train the DNN model discussed in Sect. 6 on the Google Cloud Platform with a 2.20 GHz Intel Xeon CPU and an NVIDIA Tesla K80 GPU using 16 GB of memory.

7.1 Comparison Against Existing Tools

To answer our first research question, we compare Concord against both Neo and DeepCoder on the 100 synthesis benchmarks discussed earlier. The result of this comparison is shown in Fig. 5, which plots the number of benchmarks solved within a given time limit for each of the three tools. As we can see from this figure, Concord outperforms DeepCoder and Neo both in terms of synthesis time as well as the number of benchmarks solved within the 5-min time limit. In particular, Concord can solve 82% of these benchmarks with an average running time of 36 s, whereas Neo (resp. DeepCoder) solves 71% (resp. 32%) with an average running time of 99 s (resp. 205 s). Thus, we believe these results answer our first research question in a positive way.

7.2 Ablation Study

To answer our remaining research questions, we perform an ablation study in which we compare Concord against three variants:

- **Concord-noRL:** This variant does not use reinforcement learning to update its policy during synthesis. However, it still uses the pre-trained policy to guide search, and it also uses deduction to prune infeasible partial programs. In other words, Concord-noRL is the same as the synthesis algorithm from Fig. 3 but it does not invoke the `UpdatePolicy` procedure to improve its policy during synthesis.
- **Concord-NoDeduce:** This variant uses reinforcement learning; however, it does not incorporate feedback from the deduction engine. That is, rather than checking feasibility of partial programs, it instead samples complete programs and uses the percentage of passing input-output examples as the reward signal. Note that this variant of Concord essentially corresponds to the technique proposed by Si et al. [44].[4]
- **Concord-StandardPG:** Recall that our algorithm uses an off-policy variant of the standard policy gradient algorithm to incorporate additional feedback from the deduction engine. To evaluate the benefit of our proposed approach, we created a variant called Concord-StandardPG that uses the standard (i.e., on-policy) policy gradient algorithm. In other words, ConcordStandardPG implements the same synthesis algorithm from Fig. 3 except that it uses Theorem 1 to update θ instead of Theorem 2.

The results from this evaluation are summarized in Table 1. Here, the first column labeled "# solved" shows the number of solved benchmarks, and the second column shows percentage improvement over Neo in terms of benchmarks solved. The third column shows average synthesis time for benchmarks that can

[4] We reimplement the RL algorithm proposed in [44] since we cannot directly compare against their tool. Specifically, the policy network in their implementation is tailored to their problem domain.

Table 1. Results of ablation study result comparing different variants.

	# solved	Delta to NEO	Avg. time (s)	Speedup over NEO
CONCORD-noRL	56	−21%	48	1.63×
CONCORD-NoDeduce	65	−8%	21	3.66×
CONCORD-StandardPG	65	−8%	27	2.88×
CONCORD	82	+15%	9	8.71×

be solved by *all* variants and NEO. Finally, the last column shows speed-up in terms of synthesis time compared to NEO.

As we can see from this table, all variants are significantly worse than CONCORD in terms of the number of benchmarks that can be solved within a 5-min time limit[5]. Furthermore, as we can see from the column labeled "Delta to NEO", all of our proposed ideas are important for improving over the state-of-the-art, as NEO outperforms all three variants but not the full CONCORD system, which solves 15% more benchmarks compared to NEO.

Next, looking at the third column of Table 1, we see that all three variants of CONCORD are significantly slower compared to CONCORD in terms of synthesis time. While both CONCORD and all of its variants outperform NEO in terms of synthesis time (for benchmarks solved by all tools), CONCORD by far achieves the greatest speed-up over NEO.

In summary, the results from Table 1 highlight that all of our proposed ideas (i.e., (1) improving policy at synthesis time; (2) using feedback from deduction; and (3) off-policy RL) make a significant difference in practice. Thus, we conclude that the ablation study positively answers our last three research questions.

8 Related Work

In this section, we survey prior work that is closely related to the techniques proposed in this paper.

Program Synthesis. Over the past decade, there has been significant interest in automatically synthesizing programs from high-level expressions of user intent [2,6,21,23,25,39,40,46]. Some of these techniques are geared towards computer end-users and therefore utilize informal specifications such as input-output examples [23,40,50], natural language [24,42,55,56], or a combination of both [10,12]. On the other hand, program synthesis techniques geared towards programmers often utilize additional information, such as a program sketch [17,36,46,49] or types [33,39] in addition to test cases [20,30] or logical specifications [6,49]. While the synthesis methodology proposed in this paper

[5] To understand the improvement brought by the pre-trainedd policy, we also conduct a baseline experiment by using randomly initialized policy in CONCORD. Given the setting, CONCORD can solve as many as 27% of the benchmarks in the given 5-min time limit.

can, in principle, be applied to a broad set of specifications, the particular featurization strategy we use in our implementation is tailored towards input-output examples.

Deduction-Based Pruning. In this paper, we build on a line of prior work on using deduction to prune the search space of programs in a DSL [18,19,21,39,53]. Some of these techniques utilize type-information and type-directed reasoning to detect infeasible partial programs [20–22,37,39]. On the other hand, other approaches use some form of lightweight program analysis to prune the search space [18,19,53]. Concretely, Blaze uses abstract interpretation to build a compact version space representation capturing the space of all feasible programs [53]; Morpheus [19] and Neo [18] utilize logical specifications of DSL constructs to derive specifications of partial programs and query an SMT solver to check for feasibility; Scythe [50] and Viser [51] use deductive reasoning to compute approximate results of partial programs to check their feasibility. Our approach learns from deduction feedback to improve search efficiency. As mentioned in Sect. 6, the deductive reasoning engine used in our implementation is similar to the latter category; however, it can, in principle, be used in conjunction with other deductive reasoning techniques for pruning the search space.

Learning from Failed Synthesis Attempts. The technique proposed in this paper can utilize feedback from the deduction engine in the form of other infeasible partial programs. This idea is known as *conflict-driven learning* and has been recently adopted from the SAT solving literature [5,57] to program synthesis [18]. Specifically, Neo uses the unsat core of the program's specification to derive other infeasible partial programs that share the same root cause of failure, and, as described in Sect. 6, we use the same idea in our implementation of the deduction engine. While we use logical specifications to infer other infeasible programs, there also exist other techniques (e.g., based on testing [54]) to perform this kind of inference.

Machine Learning for Synthesis. This paper is related to a long line of work on using machine learning for program synthesis. Among these techniques, some of them train a machine learning model (typically a deep neural network) to directly predict a full program from the given specification [12,16,34,35]. Many of these approaches are based on sequence-to-sequence models [47], sequence to tree models [56], or graph neural networks [41] commonly used in machine translation.

A different approach, sometimes referred to as *learning to search*, is to train a statistical model that is used to *guide* the search rather than directly predict the target program. For example, DeepCoder [2] uses a deep neural network (DNN) to predict the most promising grammar productions to use for the given input-output examples. Similarly, R3NN [38] and NGDS [26] use DNNs to predict the most promising grammar productions conditioned on both the specification and the current partial program. In addition, there has been work on using concrete program executions on the given input-output examples to guide the DNN [11,52]. Our technique for pretraining the initial policy network is based

on the same ideas as these supervised learning approaches; however, their initial policies do not change during the synthesis algorithm, whereas we continue to update the policy using RL.

While most of the work at the intersection of synthesis and machine learning uses *supervised learning* techniques, recent work has also proposed using reinforcement learning to speed up syntax-guided synthesis [8,29,31,44] These approaches are all on-policy and do not incorporate feedback from a deduction engine. In contrast, in our problem domain, rewards are very sparse in the program space, which makes exploration highly challenging in a on-policy learning setting. Our approach addresses this problem using off-policy RL to incorporate feedback from the deduction engine. Our ablation study results demonstrate that our off-policy RL is able to scale to more complex benchmarks.

Reinforcement Learning for Formal Methods. There has been recent interest in applying reinforcement learning (RL) to solve challenging PL problems where large amounts of labeled training data are too expensive to obtain. For instance, Si et al. use graph-based RL to automatically infer loop invariants [43], Singh et al. use Q-learning (a different RL algorithm) to speed up program analysis based on abstract interpretation [45], Dai et al. [14] uses meta-reinforcement learning for test data generation, and Chen et al. [9] uses RL to speed up relational program verification. However, these approaches only use RL offline to pretrain a DNN policy used to guide search. In contrast, we perform reinforcement learning online during synthesis. Bastani et al. has used an RL algorithm called Monte-carlo tree search (MCTS) to guide a specification inference algorithm [3]; however, their setting does not involve any kind of deduction.

9 Conclusion and Future Work

We presented a new program synthesis algorithm based on reinforcement learning. Given an initial policy trained off-line, our method uses this policy to guide its search at synthesis time but also gradually improves this policy using feedback obtained from a deductive reasoning engine. Specifically, we formulated program synthesis as a reinforcement learning problem and proposed a new variant of the *policy gradient* algorithm that is better suited to solve this problem. In addition, we implemented the proposed approach in a new tool called Concord and evaluated it on 100 synthesis tasks taken from prior work. Our evaluation shows that Concord outperforms a state-of-the-art tool by solving 15% more benchmarks with an average speedup of 8.71×. In addition, our ablation study highlights the advantages of our proposed reinforcement learning algorithm.

There are several avenues for future work. First, while our approach is applicable to different DSLs and specifications, our current implementation focuses on input-output examples. Thus, we are interested in extending our implementation to richer types of specifications and evaluating our method in application domains that require such specifications. Another interesting avenue for future work is to integrate our method with other types of deductive reasoning engines.

In particular, while our deduction method is based on SMT, it would be interesting to try other methods (e.g., based on types or abstract interpretation) in conjunction with our proposed RL approach.

References

1. Alur, R., et al.: Syntax-guided synthesis. IEEE (2013)
2. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: learning to write programs. In: Proceedings of International Conference on Learning Representations. OpenReview (2017)
3. Bastani, O., Sharma, R., Aiken, A., Liang, P.: Active learning of points-to specifications. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 678–692 (2018)
4. Bavishi, R., Lemieux, C., Fox, R., Sen, K., Stoica, I.: AutoPandas: neural-backed generators for program synthesis. PACMPL **3**(OOPSLA), 168:1–168:27 (2019)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, pp. 131–153 (2009)
6. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 467–481 (2017)
7. Brockschmidt, M., Allamanis, M., Gaunt, A.L., Polozov, O.: Generative code modeling with graphs. In: ICLR (2019)
8. Bunel, R., Hausknecht, M., Devlin, J., Singh, R., Kohli, P.: Leveraging grammar and reinforcement learning for neural program synthesis. In: ICLR (2018)
9. Chen, J., Wei, J., Feng, Y., Bastani, O., Dillig, I.: Relational verification using reinforcement learning. PACMPL **3**(OOPSLA), 14:11–14:130 (2019)
10. Chen, Q., Wang, X., Ye, X., Durrett, G., Dillig, I.: Multi-modal synthesis of regular expressions (2019)
11. Chen, X., Liu, C., Song, D.: Execution-guided neural program synthesis. In: ICLR (2018)
12. Chen, Y., Martins, R., Feng, Y.: Maximal multi-layer specification synthesis. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, 26–30 August 2019, pp. 602–612 (2019)
13. Cho, K., et al.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014)
14. Dai, H., Li, Y., Wang, C., Singh, R., Huang, P., Kohli, P.: Learning transferable graph exploration. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, Vancouver, BC, Canada, 8–14 December 2019, pp. 2514–2525 (2019). http://papers.nips.cc/paper/8521-learning-transferable-graph-exploration
15. 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
16. Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.R., Kohli, P.: RobustFill: neural program learning under noisy I/O. In: Proceedings of the 34th International Conference on Machine Learning, vol. 70, pp. 990–998. JMLR.org (2017)

17. Ellis, K., Ritchie, D., Solar-Lezama, A., Tenenbaum, J.: Learning to infer graphics programs from hand-drawn images. In: Advances in Neural Information Processing Systems, pp. 6059–6068 (2018)
18. Feng, Y., Martins, R., Bastani, O., Dillig, I.: Program synthesis using conflict-driven learning. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 420–435 (2018)
19. Feng, Y., Martins, R., Van Geffen, J., Dillig, I., Chaudhuri, S.: Component-based synthesis of table consolidation and transformation tasks from examples. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 422–436. ACM (2017)
20. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.: Component-based synthesis for complex APIs. In: Proceedings of Symposium on Principles of Programming Languages, pp. 599–612. ACM (2017)
21. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 229–239 (2015)
22. Frankle, J., Osera, P., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 802–815 (2016)
23. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of Symposium on Principles of Programming Languages, pp. 317–330. ACM (2011)
24. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Mapping language to code in programmatic context. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 31 October–4 November 2018, pp. 1643–1652 (2018). https://www.aclweb.org/anthology/D18-1192/
25. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proceedings of International Conference on Software Engineering, pp. 215–224. ACM/IEEE (2010)
26. Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., Gulwani, S.: Neural-guided deductive search for real-time program synthesis from examples. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, 30 April–3 May 2018, Conference Track Proceedings (2018). https://openreview.net/forum?id=rywDjg-RW
27. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp. 70–80 (2016)
28. Levine, S., Koltun, V.: Guided policy search. In: International Conference on Machine Learning, pp. 1–9 (2013)
29. Liang, C., Norouzi, M., Berant, J., Le, Q.V., Lao, N.: Memory augmented policy optimization for program synthesis and semantic parsing. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, Montréal, Canada, 3–8 December 2018, pp. 10015–10027 (2018). http://papers.nips.cc/paper/8204-memory-augmented-policy-optimization-for-program-synthesis-and-semantic-parsing
30. Long, F., Amidon, P., Rinard, M.: Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 727–739 (2017)

31. Mao, J., Gan, C., Kohli, P., Tenenbaum, J.B., Wu, J.: The neuro-symbolic concept learner: interpreting scenes, words, and sentences from natural supervision. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, 6–9 May 2019 (2019). https://openreview.net/forum?id=rJgMlhRctm
32. Martins, R., Chen, J., Chen, Y., Feng, Y., Dillig, I.: Trinity: an extensible synthesis framework for data science. Proc. VLDB Endow. **12**(12), 1914–1917 (2019)
33. Miltner, A., Maina, S., Fisher, K., Pierce, B.C., Walker, D., Zdancewic, S.: Synthesizing symmetric lenses. Proc. ACM Program. Lang. **3**(ICFP), 1–28 (2019)
34. Neelakantan, A., Le, Q.V., Abadi, M., McCallum, A., Amodei, D.: Learning a natural language interface with neural programmer. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, 24–26 April 2017, Conference Track Proceedings (2017). https://openreview.net/forum?id=ry2YOrcge
35. Neelakantan, A., Le, Q.V., Sutskever, I.: Neural programmer: inducing latent programs with gradient descent. In: 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, 2–4 May 2016, Conference Track Proceedings (2016). http://arxiv.org/abs/1511.04834
36. Nye, M.I., Hewitt, L.B., Tenenbaum, J.B., Solar-Lezama, A.: Learning to infer program sketches. In: Proceedings of the 36th International Conference on Machine Learning, ICML 2019, Long Beach, California, USA, 9–15 June 2019, pp. 4861–4870 (2019). http://proceedings.mlr.press/v97/nye19a.html
37. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 619–630 (2015)
38. Parisotto, E., Mohamed, A.R., Singh, R., Li, L., Zhou, D., Kohli, P.: Neuro-symbolic program synthesis. In: ICLR (2017)
39. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 522–538 (2016)
40. Polozov, O., Gulwani, S.: FlashMeta: a framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, Part of SPLASH 2015, Pittsburgh, PA, USA, 25–30 October 2015, pp. 107–126 (2015)
41. Shin, E.C., Allamanis, M., Brockschmidt, M., Polozov, A.: Program synthesis and semantic parsing with learned code idioms. In: Advances in Neural Information Processing Systems, pp. 10824–10834 (2019)
42. Shin, R., Allamanis, M., Brockschmidt, M., Polozov, O.: Program synthesis and semantic parsing with learned code idioms. In: NeurIPS (2019)
43. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, Montréal, Canada, 3–8 December 2018, pp. 7762–7773 (2018)
44. Si, X., Yang, Y., Dai, H., Naik, M., Song, L.: Learning a meta-solver for syntax-guided program synthesis. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, 6–9 May 2019 (2019)
45. Singh, G., Püschel, M., Vechev, M.T.: Fast numerical program analysis with reinforcement learning. In: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, 14–17 July 2018, Proceedings, Part I, pp. 211–229 (2018)

46. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, 21–25 October 2006, pp. 404–415 (2006)
47. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
48. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems, pp. 1057–1063 (2000)
49. Torlak, E., Bodík, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, 09–11 June 2014, pp. 530–541 (2014)
50. Wang, C., Cheung, A., Bodik, R.: Synthesizing highly expressive SQL queries from input-output examples. In: Proceedings of Conference on Programming Language Design and Implementation, pp. 452–466. ACM (2017)
51. Wang, C., Feng, Y., Bodík, R., Cheung, A., Dillig, I.: Visualization by example. PACMPL **4**(POPL), 49:1–49:28 (2020). https://doi.org/10.1145/3371117
52. Wang, C., Huang, P., Polozov, A., Brockschmidt, M., Singh, R.: Execution-guided neural program decoding. CoRR abs/1807.03100 (2018). http://arxiv.org/abs/1807.03100
53. Wang, X., Dillig, I., Singh, R.: Program synthesis using abstraction refinement. In: Proceedings of Symposium on Principles of Programming Languages, pp. 63:1–63:30. ACM (2018)
54. Wang, Y., Dong, J., Shah, R., Dillig, I.: Synthesizing database programs for schema refactoring. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, 22–26 June 2019, pp. 286–300 (2019)
55. Yaghmazadeh, N., Wang, Y., Dillig, I., Dillig, T.: SQLizer: query synthesis from Natural Language. In: Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 63:1–63:26. ACM (2017)
56. Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., Radev, D.: SyntaxSQLNet: syntax tree networks for complex and cross-domain text-to-SQL task. In: Proceedings of EMNLP. Association for Computational Linguistics (2018)
57. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in Boolean satisfiability solver. In: Proceedings of International Conference on Computer-Aided Design, pp. 279–285. IEEE Computer Society (2001)

Manthan: A Data-Driven Approach for Boolean Function Synthesis

Priyanka Golia[1,2(✉)], Subhajit Roy[1], and Kuldeep S. Meel[2]

[1] Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur, India
{pgolia,subhajit}@cse.iitk.ac.in

[2] School of Computing, National University of Singapore, Singapore, Singapore
meel@comp.nus.edu.sg

Abstract. Boolean functional synthesis is a fundamental problem in computer science with wide-ranging applications and has witnessed a surge of interest resulting in progressively improved techniques over the past decade. Despite intense algorithmic development, a large number of problems remain beyond the reach of the state of the art techniques.

Motivated by the progress in machine learning, we propose Manthan, a novel data-driven approach to Boolean functional synthesis. Manthan views functional synthesis as a classification problem, relying on advances in constrained sampling for data generation, and advances in automated reasoning for a novel proof-guided refinement and provable verification. On an extensive and rigorous evaluation over 609 benchmarks, we demonstrate that Manthan significantly improves upon the current state of the art, solving 356 benchmarks in comparison to 280, which is the most solved by a state of the art technique; thereby, we demonstrate an increase of 76 benchmarks over the current state of the art. Furthermore, Manthan solves 60 benchmarks that none of the current state of the art techniques could solve. The significant performance improvements, along with our detailed analysis, highlights several interesting avenues of future work at the intersection of machine learning, constrained sampling, and automated reasoning.

1 Introduction

Given an existentially quantified Boolean formula $\exists Y F(X,Y)$ over the set of variables X and Y, the problem of Boolean functional synthesis is to compute a vector of Boolean functions, denoted by $\Psi(X) = \langle \psi_1(X), \psi_2(X), \ldots, \psi_{|Y|}(X) \rangle$, and referred to as Skolem function vector, such that $\exists Y F(X,Y) \equiv F(X, \Psi(X))$. In the context of applications, the sets X and Y are viewed as inputs and outputs, and the formula $F(X,Y)$ is viewed as a functional specification capturing the relationship between X and Y, while the Skolem function vector $\Psi(X)$ allows one to determine the value of Y for the given X by evaluating Ψ. The study of

The open source tool is available at https://github.com/meelgroup/manthan.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 611–633, 2020.
https://doi.org/10.1007/978-3-030-53291-8_31

Boolean functional synthesis traces back to Boole [12], and over the decades, the problem has found applications in a wide variety of domains such as certified QBF solving [8,9,36,41], automated program repair [27], program synthesis [44], and cryptography [35].

Theoretical investigations have demonstrated that there exist instances where Boolean functional synthesis takes super-polynomial time. On the other hand, practical applicability has necessitated the development of algorithms with progressively impressive scaling. The algorithmic progress for Boolean functional synthesis has been driven by a diverse set of techniques: (i) the usage of incremental determinization employing the several heuristics in state-of-the-art Conflict Driven Clause Learning (CDCL) solvers [41], (ii) usage of decomposition techniques employing the progress in knowledge compilation [6,19,28,45], and (iii) Counter-Example Guided Abstraction Refinement (CEGAR)-based techniques relying on usage of SAT solvers as black boxes [4–6,28]. While the state of the art techniques are capable of handling problems of complexity beyond the capability of tools a decade ago, the design of scalable algorithms capable of handling industrial problems remains the holy grail.

In this work, we take a step towards the above goal by proposing a novel approach, called Manthan, at the intersection of machine learning, constrained sampling, and automated reasoning. Motivated by the unprecedented advances in machine learning, we view the problem of functional synthesis through the lens of multi-class classification aided by the generation of the data via constrained sampling and employ automated reasoning to certify and refine the learned functions. To this end, the architecture of Manthan comprises of the following three novel techniques:

Data Generation. The state of the art machine learning techniques use training data represented as a set of samples where each sample consists of valuations to features and the corresponding label. In our context, we treat X as the features and Y as labels. Unlike the standard setup of machine learning wherein for each assignment to X, there is a unique label, i.e. assignment to Y, the relationship between X and Y is captured by a relation and not necessarily a function. To this end, we design a weighted sampling strategy to generate a *representative* data set that can be fitted using a *compactly sized* classifier. The weighted sampling strategy, implemented using state of the constrained sampler, seeks to uniformly sample input variables (X) while biasing the valuations of output variables towards a particular value.

Dependency-Driven Classifier for Candidates. Given training data viewed as a valuation of *features* (X) and their corresponding labels (Y), a natural approach from machine learning perspective would be to perform multi-class classification to obtain $Y = h(X)$, where h is a symbolic representation of the learned classifier. Such an approach, however, can not ensure that h can be expressed as a vector of Boolean functions. To this end, we design a dependency aware classifier to construct a vector of decision trees corresponding to each Y_i, wherein each decision tree is expressed as a Boolean function.

Proof-Guided Refinement. Since machine learning techniques often produce good but inexact approximations, we augment our method with automated reasoning techniques to verify the correctness of decision tree-based candidate Skolem functions. To this end, we perform a counterexample driven refinement approach for candidate Skolem functions.

To fully utilize the impressive test accuracy attained by machine learning models, we design a *proof-guided refinement* approach that seeks to identify and apply *minor* repairs to the candidate functions, in an iterative manner, until we converge to a provably correct Skolem function vector. In a departure from prior approaches utilizing the Shannon expansion and self-substitution, we first use a MaxSAT solver to determine potential repair candidates, and employ unsatisfiability cores obtained from the infeasibility proofs capturing the reason for current candidate functions to meet the specification, to construct a *good repair*.

Finally, We perform an extensive evaluation over a diverse set of benchmarks with state-of-the-art tools, viz. C2Syn [4], BFSS [5], and CADET [39]. Of 609 benchmarks, Manthan is able to solve 356 benchmarks while C2Syn, BFSS, and CADET solve 206, 247, and 280 benchmarks respectively. Significantly, Manthan can solve 60 benchmarks beyond the reach of all the other existing tools extending the reach of functional synthesis tools. We then perform an extensive empirical evaluation to understand the impact of different design choices on the performance of Manthan. Our study reveals several surprising observations arising from the inter-play of machine learning and automated reasoning.

Manthan owes its runtime performance to recent advances in machine learning, constrained sampling, and automated reasoning. Encouraged by Manthan's scalability, we will seek to extend the above approach to related problem domains such as automated program synthesis, program repair, and reactive synthesis.

The rest of the paper is organized as follows: We first introduce notations and preliminaries in Sect. 2. We then discuss the related work in Sect. 3. In Sect. 4 we present an overview of Manthan and give an algorithmic description in Section 5. We then describe the experimental methodology and discuss results in Sect. 6. Finally, we conclude in Sect. 7.

2 Notations and Preliminaries

We use lower case letters (with subscripts) to denote propositional variables and upper case letters to denote a subset of variables. The formula $\exists Y F(X, Y)$ is existentially quantified in Y, where $X = \{x_1, \cdots, x_n\}$ and $Y = \{y_1, \cdots, y_m\}$. For notational clarity, we use F to refer to $F(X, Y)$ when clear from the context. We denote $Vars(F)$ as the set of variables appearing in $F(X, Y)$. A literal is a boolean variable or its negation. We often abbreviate universally (resp. existentially) quantified variables as universal (resp. existential) variables.

A *satisfying assignment* of a formula $F(X, Y)$ is a mapping $\sigma : Vars(F) \to \{0, 1\}$, on which the formula evaluates to True. For $V \subseteq Vars(F)$, $\sigma[V]$ represents the truth values of variables in V in a satisfying assignment σ of F.

We denote the set of all witnesses of F as R_F. For a formula in conjunctive normal form, the *unsatisfiable core*(UnsatCore) is a subset of clauses of the formula for which no satisfying assignment exists.

We use $F(X,Y)|_{y_i=b}$ to denote *substitutions*: a formula obtained after substituting every occurrence of y_i in $F(X,Y)$ by b, where b can be a constant (0 or 1) or a formula. The operator *ite(condition,exp1,exp2)* is used to represent the if-else case: if the *condition* is true, then it returns *exp1*, else it returns *exp2*.

A variable y_i is considered as a *positive unate* if and only if $F(X,Y)|_{y_i=0} \wedge \neg F(X,Y)|_{y_i=1}$ is UNSAT and a *negative unate* if and only if $F(X,Y)|_{y_i=1} \wedge \neg F(X,Y)|_{y_i=0}$ is UNSAT [5].

Given a function vector $\langle \psi_1, \ldots, \psi_m \rangle$ for the vector of variables $\langle y_1, \ldots y_m \rangle$ such that ψ_i is the function corresponding to y_i, we say that there exists a partial order $\prec_d$ over the variables $\{y_1, \ldots y_m\}$ such that $y_i \prec_d y_j$ if ψ_i depends on y_j.

In decision tree learning, a fraction of incorrectly assigned labels refer to the *impurity*. We use Gini Index [38] as a measure of *impurity* for a class label. The *impurity decrease* at a node is the difference of its impurity to the mean of impurities of its children. The *minimum impurity decrease* is a hyper-parameter used to control the maximum allowable impurity at the leaf nodes, thereby providing a lever for how closely the classifier fits the training data.

Given a propositional formula $F(X,Y)$ and a weight function $W(\cdot)$ assigning non-negative weights to every literal, we refer to the *weight* of a satisfying assignment σ, denoted as $W(\sigma)$, as the product of weights of all the literals appearing in σ, i.e., $W(\sigma) = \prod_{l \in \sigma} W(l)$. A *sampler* $\mathcal{A}(\cdot,\cdot)$ is a probabilistic generator that guarantees $\forall \sigma \in R_F$, $\mathsf{Pr}\left[\mathcal{A}(F, \mathsf{Bias}) = \sigma\right] \propto W(\sigma)$.

We use a function Bias that takes a mapping from a sequence of variables to the desired weights of their positive literals, and assigns corresponding weights to each of the positive literals. We use a simpler notation, `Bias(a,b)` to denote that positive literals corresponding to all universal variables are assigned a weight `a` and positive literals corresponding to all existential variables are assigned a weight `b`. For example, Bias(0.5, 0.9) assigns a weight of 0.5 to the positive literals of the universally quantified variables and 0.9 to the positive literals of the existentially quantified variables.

Problem Statement: Given a Boolean specification $F(X,Y)$ between set of inputs $X = \{x_1, \cdots, x_n\}$ and vector of outputs $Y = \langle y_1, \cdots, y_m \rangle$, the problem of *Skolem function synthesis* is to synthesize a function vector $\Psi = \langle \psi_1(X), \cdots, \psi_m(X) \rangle$ such that $y_i \leftrightarrow \psi_i(X)$ and $\exists Y F(X,Y) \equiv F(X,\Psi)$. We refer to Ψ as the *Skolem function vector* and ψ_i as the *Skolem function* for y_i.

A variable y_i is called self-substituted variable, if the Skolem function ψ_i corresponding to y_i is set to $F(X,Y)|_{y_i=1}$ [19].

Given a formula $\exists Y F(X,Y)$ and a Skolem function vector Ψ, we refer to $E(X,Y,Y')$ as an *error formula* [28], where $Y' = \{y'_1, \cdots, y'_{|Y|}\}$, and $Y' \neq Y$.

$$E(X,Y,Y') = F(X,Y) \wedge \neg F(X,Y') \wedge (Y' \leftrightarrow \Psi) \tag{1}$$

We use the following theorems from prior work:

Theorem 1 ([28]). *Ψ is a Skolem function if and only if $E(X, Y, Y')$ is UNSAT.*

Theorem 2 ([5]). *If y_i is positive(resp negative) unate in $F(X, Y)$, then $\psi_i = 1$ (resp $\psi_i = 0$) is the Skolem function for y_i.*

3 Related Work

The origins of the problem of Boolean functional synthesis traces back to Boole's seminal work [12], which was subsequently rigorously pursued, albeit focused on decidability, by Lowenheim and Skolem [33]. The complexity theoretic studies have shown that there exist instances where Boolean functional synthesis takes super polynomial time and was also shown that there exist instances for which polynomial size Skolem function vector does not suffice unless Polynomial Hierarchy (PH) collapses [5].

Motivated by the success of the CEGAR (Counter-Example Guided Abstraction Refinement) approach in model checking, CEGAR-based approaches have been pursued in the context of synthesis as well, where the key idea is to use a Conflict-Driven Clause Learning (CDCL) SAT solver to verify and refine the candidate Skolem functions [4–6,28].

Another line of work has focused on the representation of specification, i.e., $F(X, Y)$, in representations that are amenable to efficient synthesis for a class of functions. The early approaches focused on ROBDD representation building on the functional composition approach proposed by Balabanov and Jiang [8]. Building on Tabajara and Vardi's ROBDD-based approach [45], Chakraborty et al. extended the approach to factored specifications [14]. It is worth mentioning that factored specifications had earlier been pursued in the context of CEGAR-based approaches. Motivated by the success of knowledge compilation in the field of probabilistic reasoning, Akshay et al. achieved a significant breakthrough over a series of papers [5,6,28] to propose a new negation normal form, SynNNF [4]. The generalization and a functional specification presented in SynNNF is amenable to efficient functional synthesis [4]. Another line of work focused on the usage of *incremental determinization* to incrementally construct the Skolem functions [25,30,36,39,41].

Several approaches have been proposed for the particular case when the specification, $\exists Y F(X, Y)$ is valid, i.e., $\forall X \exists Y F(X, Y)$ is True. Inspired by the sequential relational decomposition, Chakraborty et al. [14] recently proposed an approach focused on viewing each CNF clause of the specification consisting of *input and output* clauses and employing a *cooperation*-based strategy. The progress in modern CDCL solvers has led to an exploration of usage of heuristics for problems in complexity classes beyond NP. This has led to work on the extraction of Skolem functions from the proofs constructed for the formulas expressed as $\forall X \exists Y F(X, Y)$ [8,9].

The performance of Manthan crucially depends on its ability to employ constrained sampling, which has witnessed a surge of interest with approaches

ranging from those based on hashing-based techniques [15], knowledge compilation [24,42], augmentation of SAT solvers with heuristics [43].

The recent success of machine learning has led to several attempts to the usage of machine learning in several related synthesis domains such as program synthesis [7], invariant generation, decision-tree for functions in Linear Integer Arithmetic theory using pre-specified examples [18], strategy synthesis for QBF [26]. Use of data-driven approaches for invariant synthesis has been investigated in the ICE learning framework [17,20,21] aimed with data about the program behavior from test executions, it proposes invariants by learning from data, checks for inductiveness and, on failure, extend the data by the generated counterexamples. The usage of proof-artifacts such as unsat cores has been explored in verification since early 2000s [23] and in program repair in Wolverine [46], while MaxSAT has been used in program debugging in [10,29].

4 Manthan: An overview

In this section, we provide an overview of our proposed framework, Manthan, before divulging into core algorithmic details in the following section. Manthan takes in a function specification, represented as $F(X,Y)$, and returns a Skolem function vector $\Psi(X)$ such that $\exists Y F(X,Y) \equiv F(X,\Psi(X))$. As shown in Fig. 1 Manthan consists of following three phases:

1. Preprocess employs state-of-the-art pre-processing techniques on F to compute a partial Skolem function vector.
2. LearnSkF takes in the pre-processed formula and uses constrained samplers, and classification techniques to compute candidate Skolem functions for all the variables in Y.
3. Refine performs verification and proof-guided refinement procedure wherein a SAT solver is employed to verify the correctness of candidate functions and a MaxSAT solver in conjunction with a SAT solver is employed to refine the candidate functions until the entire candidate Skolem function vector passes the verification check.

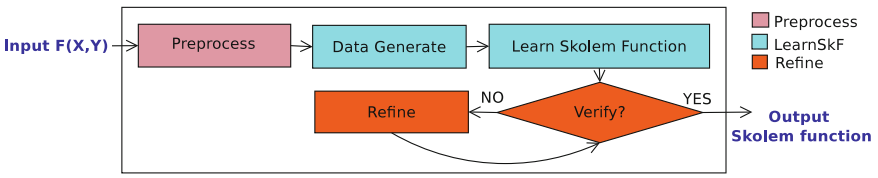


Fig. 1. Overview of Manthan

We now provide a high-level description of different phases to highlight the technical challenges, which provides context for several algorithmic design choices presented in the next section.

4.1 Phase 1: Preprocess

Preprocess focuses on pre-processing of the formula to search for unates among the variables in Y; if y_i is positive (resp. negative) unate, then $\psi_i = 1$(resp. 0) suffices as a Skolem function. We employ the algorithmic routine proposed by Akshay et al. [5] to drive this preprocessing.

4.2 Phase 2: LearnSkF

LearnSkF views the problem of functional synthesis through the lens of machine learning where the learned machine learning model for classification of a variable y_i can be viewed as a candidate Skolem function for y_i. We gather training data about the function's behavior by exploiting the progress in constrained sampling to sample solutions of $F(X, Y)$. Recall that $F(X, Y)$ defines a relation (and not necessarily a function) between X and Y, and the machine learning techniques typically assume the existence of function between features and labels, necessitating the need for sophisticated sampling strategy as discussed below. Moving on to features and labels, since we want to learn Y in terms of X, we view X as a set of features while assignments to Y as a set of class labels.

The off-the-shelf classification techniques typically require that the size of training data is several times larger than the size of possible class labels, which would be prohibitively large for the typical problems involving more than thousand variables. To mitigate the requirement of large training data, we make note of two well-known observations in functional synthesis literature: (1) the Skolem function ψ_i for a variable y_i typically does not depend on all the variables in X, (2) A Skolem function vector Ψ where ψ_i depends on variable y_j is a valid vector if the Skolem function ψ_j is not dependent on y_i (i.e., acyclic dependency), i.e., there exists a partial order $\prec_d$ over $\{y_1, \ldots y_m\}$.

The above observations lead us to design an algorithmic procedure where we learn candidate Skolem functions as decision trees in an iterative manner, i.e., one y_i at a time, thereby allowing us to constrain ourselves to the binary classification. The learned classifier can then be represented as the disjunction of all the paths from the root to the leaves in the learnt decision tree. We update the set of possible features for a given y_i depending on the candidate functions generated so far, i.e., valuation of X variables and Y variables, which are not dependent on y_i. Finally, we compute the candidate Skolem function for y_i as the disjunction of labels along edges for all the paths from the root to leaf nodes with label 1. Once, we have the candidate Skolem function vector Ψ, we obtain a valid linear extension, *TotalOrder*, of the partial order $\prec_d$ in accordance to Ψ.

Before moving on to the next phase, we return to the formulation of sampling. The past few years have witnessed the design of uniform [15,42], and weighted samplers [24], and one wonders what kind of sampler should we choose to generate samples for training data. A straightforward choice would be to perform uniform sampling over X and Y, but the relational nature of specification, F, between X and Y offers interesting challenges and opportunities. Recall while F specifies a relation between X and Y, we are interested in a Skolem function, and

we would like to tailor our sampling subroutines to allow discovery of Skolem functions with *small* description given the relationship between description and sample complexity. To this end, consider $X = \{x_1, x_2\}$ and $Y = \{y_1\}$, and let $F := (x_1 \vee x_2 \vee y_1)$. Note that F has 7 solutions over $X \cup Y$, out of which $y_1 = 0$ appears in 3 solutions while $y_1 = 1$ appears in 4. Also, note that there are several possible Skolem functions such as $y_1 = \neg(x_1 \wedge x_2)$. Now, if we uniformly sample solutions of F over x_1, x_2, y_1, i.e. Bias(0.5, 0.5), we would see (almost) equal number of samples with $y_1 = 0$ and $y_1 = 1$. A closer look at F reveals that it is possible to construct a Skolem function by knowing that the only case where y_1 cannot be assigned 0 is when $x_1 = x_2 = 0$. To encode this intuition, we propose a novel idea of collecting samples with weighted sampling, i.e., Bias(0.5, q) where q is chosen in a multi-step process of first drawing a small set of samples with both $q = 0.9$ and $q = 0.1$, and then drawing rest of the samples by fixing the value of q following analysis of an initial set of samples. To the best of our knowledge, this is the first application of weighted sampling in the context of synthesis, and our experimental results point to several interesting avenues of future work.

4.3 Phase 3: Refine

The candidate Skolem functions generated in LearnSkF may not always be the actual Skolem functions. Hence, we require a *verification* check to see if candidate Skolem functions are indeed correct; if not, the generated counterexample can be used to *repair* it. The verification query constructs an *error formula* $E(X, Y, Y')$ (Formula 1): if unsatisfiable, the candidate Skolem function vector is indeed a Skolem function vector and the procedure can terminate; else, when $E(X, Y, Y')$ is SAT, the solution of $E(X, Y, Y')$ is used to identify and refine the erring functions among the candidate Skolem function vector.

In contrast to prior techniques that apply Shannon expansion or self-substitution, the refinement strategy in Manthan is guided by the view that the candidate function vector from the LearnSkF phase is *almost correct*, and hence, attempts to identify and apply a series of *minor* repairs to the erring functions to arrive at the correct Skolem function vector. To this end, Manthan uses two key techniques: *fault localization* and *repair synthesis*. Let us assume that σ is a satisfying assignment of $E(X, Y, Y')$ and referred to as counterexample for the current candidate Skolem function vector Ψ.

Fault Localization. In order to identify the initial candidates to repair for the counterexample σ, Manthan attempts to identify a small number of Skolem functions (correspondingly Y variables) whose outputs must undergo a change for the formula to behave correctly on σ; in other words, it makes a best-effort attempt to ensure that most of the Skolem functions (correspondingly Y variables) can retain their current output on σ while satisfying the formula. Manthan encodes this problem as a partial MaxSAT query with $F(X, Y) \wedge (X \leftrightarrow \sigma[X])$ as a hard constraint and $(Y \leftrightarrow \sigma[Y'])$ as soft constraints. All Y variables whose valuation constraint $(Y \leftrightarrow \sigma[Y'])$ does not hold in the MaxSAT solution are identified as erring Skolem functions that may need to be repaired.

Repair Synthesis. Let y_k be the variable corresponding to the erring function, ψ_k, identified in the previous step. To synthesize a repair for the function, Manthan applies a proof-guided strategy: it constructs a formula $G_k(X, Y)$, such that if $G_k(X, Y)$ is unsatisfiable then ψ_k must undergo a change. The UnsatCore of $G_k(X, Y)$ provides a *reason* that explains the discrepancy between the specification and the current Skolem function.

$$G_k(X,Y) = (y_k \leftrightarrow \sigma[y'_k]) \wedge F(X,Y) \wedge (X \leftrightarrow \sigma[X]) \wedge (\hat{Y} \leftrightarrow \sigma[\hat{Y}])$$
$$\text{where } \hat{Y} \subset Y \text{ and } \hat{Y} = \{TotalOrder[index(y_k)+1], \cdots, TotalOrder[|Y|]\} \quad (2)$$

Manthan uses the UnsatCore to constructs a *repair formula*, say β, as a conjunction over literals in the unsatisfiable core; if ψ_k is *true* with the current valuation of X and $\hat{Y}$, Manthan updates the function ψ_k by conjoining it with the negation of repair formula ($\psi_k \leftarrow \psi_k \wedge \neg\beta$); otherwise, Manthan updates the function ψ_k, by disjoining it with the repair formula ($\psi_k \leftarrow \psi_k \vee \beta$).

Self-substitution for Poorly Learnt Functions. Some Skolem functions are difficult to learn through data. In our implementation, the corresponding variables escape the LearnSkF phase with poor candidate functions, thereby requiring a long sequence of incremental repairs for convergence. To handle such scenarios, we make the following observation: though synthesizing Skolem functions via self-substitution [19] can lead to an exponential blowup in the worst case, it is inexpensive if the number of variables synthesized via this technique is small. We use this observation to quickly synthesize a Skolem function for an erring variable if we detect its candidate function is poor (detected by comparing the number of times it enters refinement against an empirically determined threshold). Of course, this heuristic does not scale well if the number of such variables is large; in our experiments, we found less than 20% of the instances solved required self-substitution, and for over 75% of these instances, only one variable needed self-substitution. We elaborate more on the empirical evidence on the success of this heuristic in Sect. 6. A theoretical understanding of the learnability of Boolean functions from data seems to be an interesting direction for future work.

5 Manthan: Algorithmic Description

In this section, we present a detailed algorithmic description of Manthan, whose pseudocode is presented in Algorithm 1. Manthan takes in a formula $F(X, Y)$ as input and returns a Skolem vector Ψ. The algorithm starts off by preprocessing (line 1) the formula $F(X, Y)$ to get the unates (U) and their corresponding Skolem functions (Ψ). Next, it invokes the sampler (line 2) to collect a set of samples(Σ) as training data for the learning phase.

For each of the existential variables that are not unates, Manthan attempts to learn candidate Skolem functions (lines 4–5). To generate a variable order,

Algorithm 1: Manthan($F(X,Y)$)

```
1  Ψ, U ← Preprocess(F(X,Y))
2  Σ ← GetSamples(F(X,Y))
3  D ← ∅
4  foreach y_j ∈ Y \ U do
5  |  ψ_j, D ← CandidateSkF(Σ, F(X,Y), y_j, D)
6  TotalOrder ← FindOrder(D)
7  repeat
8  |  E(X,Y,Y') ← F(X,Y) ∧ ¬F(X,Y') ∧ (Y' ↔ Ψ)
9  |  ret, σ ← CheckSat(E(X,Y,Y'))
10 |  if ret = SAT then
11 |  |  Ψ ← RefineSkF(F(X,Y), Ψ, σ, TotalOrder)
12 until ret = UNSAT
13 Ψ ← Substitute(F(X,Y), Ψ, TotalOrder)
14 return Ψ
```

CandidateSkF uses a collection of sets $d_1, \cdots, d_{|Y|} \in D$, such that $y_i \in d_j$ indicates that y_j depends on y_i. Next, the FindOrder routine (line 6) construct *TotalOrder* of the Y variables in accordance to the dependencies in D. The verification and refinement phase (line 8) commences by constructing the error formula and launching the verification check (line 9). If the error formula is satisfiable, the counterexample model (σ) is used to refine the formula. Once the verification check is successful, the refinement phase ends and the subroutine Substitute is invoked to recursively substitute all $y_i \in Y$ appearing in Skolem functions with their corresponding Skolem functions such that only X variables entirely describe all Skolem functions. The strict variable ordering enforced above ensures that Substitute always succeeds and does not get stuck in a cycle. Finally, the Skolem function vector Ψ is returned.

It is worth noting that Manthan can successfully solve an instance without having to necessarily execute all the phases. In particular, if $U = Y$, then Manthan terminates after Preprocess (i.e., line 1). Similarly, if the CheckSat return UNSAT during the first iteration of loop (lines 8–11), then Manthan does not invoke RefineSkF.

We now discuss each subroutine in detail. The pseudocode for Preprocess, GetSamples and Substitute is deferred to technical report [22].

Preprocess: We perform the pre-processing step as described in [5], which performs SAT queries on the formulas constructed as specified in Theorem 2.

GetSamples: GetSamples takes $F(X,Y)$ as input and returns a subset of satisfying assignments of $F(X,Y)$. GetSamples first generates a small set of samples (500) with Bias$(0.5, 0.9)$ and calculates m_i for all y_i, m_i is a ratio of number of samples with y_i being 1 to the total number of samples. Similarity, GetSamples generates 500 samples with Bias$(0.5, 0.1)$ and calculates n_i for all y_i, n_i is a ratio

Algorithm 2: CandidateSkF$(\Sigma, F(X,Y), y_j, D)$

```
1  featset ← X
2  foreach y_k ∈ Y \ y_j do
3  |  if y_j ∉ d_k then
4  |  |  featset ← featset ∪ y_k /* if y_k is not dependent on y_j */
5  feat, lbl ← Σ_{↓featset}, Σ_{↓y_j}
6  t ← CreateDecisionTree(feat, lbl)
7  foreach n ∈ LeafNodes(t) do
8  |  if Label(n) = 1 then
9  |  |  π ← Path(t, root, n)
10 |  |  ψ_j ← ψ_j ∨ π
11 foreach y_k ∈ ψ_j do
12 |  d_j ← d_j ∪ y_k ∪ d_k
13 return ψ_j, D
```

of number of samples with y_i being 0 to the total number of samples. Finally, GetSamples generates required number of samples with Bias$(0.5, q)$; for a y_i, q is m_i if both m_i and n_i are in range 0.35 to 0.65, else q is 0.9.

CandidateSkF: CandidateSkF, presented in Algorithm 2, assumes access to following three subroutines:

1. CreateDecisionTree takes the feature and label sets as input (training data) and returns a decision tree t. We use the ID3 algorithm [38] to construct a decision tree t where the internal node of t represents a feature on which a decision is made, the branches represent partitioning of the training data on the decision, and the leaf nodes represent the classification outcomes (i.e class labels). The ID3 algorithm iterates over the training data, and in each iteration, it selects a new attribute to extend the tree by a new decision node: the selected attribute is one that causes the maximum drop in the impurity of the resulting classes; we use Gini Index [38] as the measure of impurity. The algorithm, then, extends the tree by the selected decision and continues extending building the tree. The algorithm terminates on a path if either it exhausts all attributes for decisions, or the impurity of the resulting classes drop below a (user-specified) impurity decrease parameter.
2. Label takes a leaf node of the decision tree as input and returns the class label corresponding to the node.
3. Path takes a tree t and two nodes of t (node a and node b) as input and outputs a conjunction of literals in the path from node a to node b in t.

As we seek to learn Boolean functions, we employ binary classifiers with class labels 0 and 1. CandidateSkF shows our algorithm for extracting a Boolean function from the decision trees: lines 2–4 find a feature set (*featset*) to predict y_j. The feature set includes all X variables and the subset of Y variables that are not dependent on y_j. CandidateSkF creates decision tree t using samples Σ over

the feature set. Lines 7–10 generate candidate Skolem function ψ_j by iterating over all the leaf nodes of t. In particular, if a leaf node is labeled with 1, the candidate function is updated by disjoining with the formula returned by subroutine Path. CandidateSkF also updates d_j in D, d_j is set of all Y variables on which, y_j depends. If y_j depends on y_k, then by transitivity y_j also depends on d_k; in line 12, CandidateSkF updates d_j accordingly.

FindOrder: FindOrder takes D as an input to output a valid linear extension of the partial order $\prec_d$ defined over $\{y_1, \ldots y_m\}$ with respect to the candidate Skolem function vector Ψ.

Algorithm 3: RefineSkF$(F(X,Y), \Psi, \sigma, \mathit{TotalOrder})$

```
1  H ← F(X,Y) ∧ (X ↔ σ[X]); S ← (Y ↔ σ[Y'])
2  Ind ← MaxSATList(H, S)
3  foreach y_k ∈ Ind do
4  |  Ŷ ← {TotalOrder[index(y_k) + 1], ··· , TotalOrder[|Y|]}
5  |  if CheckSubstitute(y_k) then
6  |  |  ψ_k ← DoSelfSubstitution(F(X,Y), y_k, Y \ Ŷ)
7  |  else
8  |  |  G_k ← (y_k ↔ σ[y'_k]) ∧ F(X,Y) ∧ (X ↔ σ[X]) ∧ (Ŷ ↔ σ[Ŷ])
9  |  |  ret, ρ ← CheckSat(G_k)
10 |  |  if ret = UNSAT then
11 |  |  |  C ← FindCore(G_k)
12 |  |  |  β ← ⋀_{l∈C} ite((σ[l] = 1), l, ¬l)
13 |  |  |  ψ_k ← ite((σ[y'_k] = 1), ψ_k ∧ ¬β, ψ_k ∨ β)
14 |  |  else
15 |  |  |  foreach y_t ∈ Y \ Ŷ do
16 |  |  |  |  if ρ[y_t] ≠ σ[y'_t] then
17 |  |  |  |  |  Ind ← Ind.Append(y_t)
18 |  |  |  σ[y_k] ← σ[y'_k]
19 return Ψ
```

RefineSkF: RefineSkF is invoked with a counterexample σ. RefineSkF first performs *fault localization* to find the initial set of erring candidate functions; to this end, it calls the MaxSATList subroutine (line 2) with $F(X,Y) \wedge (X \leftrightarrow \sigma[X])$ as hard-constraints and $(Y \leftrightarrow \sigma[Y])$ as soft-constraints. MaxSATList employs a MaxSAT solver to find the solution that satisfies all the hard constraints and maximizes the number of satisfied soft constraints, and then returns a list (Ind) of Y variables such that for each of the variables appearing in (Ind) the corresponding soft-constraint was not satisfied by the optimal solution returned by MaxSAT solver.

Since candidate Skolem function corresponding to the variables in *Ind* needs to refine, RefineSkF now attempts to synthesize a repair for each of these candidate Skolem functions. Repair synthesis loop (lines 3–19) starts off by collecting the set of Y variables, $\hat{Y}$, on which y_k of *Ind* can depend on as per the ordering constraints (line 4). Next, it invokes the subroutine CheckSubstitute, which returns True if the candidate function corresponding to y_k has been refined more than a chosen threshold times (fixed to 10 in our implementation), and the corresponding decision tree constructed during execution CandidateSkF has exactly one node. If CheckSubstitute returns true, RefineSkF calls DoSelfSubstitution to perform self-substitution. DoSelfSubstitution takes a formula $F(X, Y)$, an existentially quantified variable y_k and a list of variables which depends on y_k and performs self substitution of y_k with constant 1 in the formula $F(X, Y)$ [28].

If CheckSubstitute returns false, RefineSkF attempts a proof-guided repair for y_k. RefineSkF calls CheckSat in line 9 on G_k, which corresponds to formula 2: if G_k is SAT, then CheckSat returns a satisfying assignment(ρ) of G_k in σ, else CheckSat returns unsatisfiable in the result, *ret*.

1. If *ret* is UNSAT, we proceed to refine ψ_k such that for $\psi_k(X \mapsto \sigma[X], \hat{Y} \mapsto \sigma[\hat{Y}]) = \sigma[y_k]$. Ideally, we would like to apply a refinement that generalizes to potentially other counter-examples, i.e. solutions of $E(X, Y, Y')$. To this end, RefineSkF calls FindCore with G_k; FindCore returns the list of variables (C) that occur in the clauses of UnsatCore of G_k. Accordingly, the algorithm constructs a *repair formula* β as a conjunction of literals in σ corresponding to variables in C (line 12). If $\sigma[y'_k]$ is 1, then ψ_k is ψ_k with conjunction of negation of β and if $\sigma[y'_k]$ is 0, then ψ_k is ψ_k with disjunction of β.
2. If *ret* is SAT and ρ is a satisfying assignment of G_k, then there exists a Skolem function vector such that the value of ψ_k agrees with $\sigma[y_k]$ for the valuation of X and $\hat{Y}$ set to $\sigma[X]$ and $\sigma[\hat{Y}]$. However, for any $y_t \in Y \setminus \hat{Y}$ if $\sigma[y'_t] \neq \rho[y'_t]$, then for such a y_t, the Skolem function corresponding to y_t may need to refine . Therefore, RefineSkF adds y_t to list of candidates to refine, *Ind*. Note that since $\sigma \models E(X, Y, Y')$, there exists at least one iteration of the loop (lines 3–18) where *ret* is UNSAT.

Substitute: To return the Skolem functions in terms of only X, Manthan invokes Substitute subroutine. For each y_j of Y variable, Substitute consider Y variables that occurs later in *TotalOrder* as $\hat{Y}$. Then, for each y_i of $\hat{Y}$; it substitutes corresponding Skolem function ψ_i in the Skolem function ψ_j of y_j.

An example to illustrate our algorithm is deferred to the technical report [22].

6 Experimental Results

We evaluate the performance of Manthan on the union of all the benchmarks employed in the most recent works [4,5],which includes 609 benchmarks from different sources: Prenex-2QBF track of QBFEval-17 [2], QBFEval-18 [3], disjunctive [6], arithmetic [45] and factorization [6]. We ran all the tools as per

the specification laid out by their authors. We used Open-WBO [34] for our MaxSAT queries and PicoSAT [11] to compute UnsatCore. We used PicoSAT for its ease of usage and we expect further performance improvements by upgrading to one of the state of the art SAT solvers. We have used the Scikit-Learn [37] to create decision trees in LearnSkF phase of Manthan. We have also used ABC [31] to represent and manipulate Boolean functions. To allow for the input formats supported by the different tools, we use the utility scripts available with the BFSS distribution [5] to convert each of the instances to both QDIMACS and Verilog formats. For Manthan, unless otherwise specified, we set the number of samples according to heuristic based on $|Y|$ as described in Sect. 6.3 and minimum impurity decrease to 0.005. All our experiments were conducted on a high-performance computer cluster with each node consisting of a E5-2690 v3 CPU with 24 cores and 96 GB of RAM, with a memory limit set to 4 GB per core. All tools were run in a single-threaded mode on a single core with a timeout of 7200 s.

The objective of our experimental evaluation was two-fold: to understand the impact of various design choices on the runtime performance of Manthan and to perform an extensive comparison of runtime performance vis-a-vis state of the art synthesis tools. In particular, we sought to answer the following questions:

1. How does the performance of Manthan compare with state of the functional synthesis engines?
2. How do the usage of different sampling schemes and the quality of samplers impact the performance of Manthan?
3. What is the impact of LearnSkF on the performance of Manthan?
4. What is the distribution of the time spent in each phase of Manthan?
5. How does using MaxSAT solver to identify the potential erring Skolem functions impacts on the performance of Manthan?
6. How does employing self-substitution for some Skolem functions impact Manthan?

We observe that Manthan significantly improves upon state of the art, and solves 356 benchmarks while the state of the art tool can only solve 280; in particular, Manthan solves 60 more benchmarks that could not be solved by any of the state of the art tools. To put the runtime performance statistics in a broader context, the number of benchmarks solved by techniques developed over the past five years range from 206 to 280, i.e., a difference of 74, which is same as an increase of 76 (i.e., from 280 to 356) due to Manthan.

Our experimental evaluation leads to interesting conclusions and several directions for future work. We observe that the performance of Manthan is sensitive to different sampling schemes and the underlying samplers; in fact, we found that biased sampling yields better results than uniform sampling. This raises interesting questions on the possibility of designing specialized samplers for this task. Similarly, we observe interesting trade offs between the number of samples and the minimum impurity decrease in LearnSkF. The diversity of our extensive benchmark suite produces a nuanced picture with respect to time distribution across different phases, highlighting the critical nature of each of the

phases to the performance of Manthan. Manthan shows significant performance improvement by using MaxSAT solver to identify candidates to refine. Manthan also has significant performance improvement with self substitution in terms of the required number of refinements.

6.1 Comparison with Other Tools

We now present performance comparison of Manthan with the current state of the art synthesis tools, BFSS [5], C2Syn [4], BaFSyn [14] and the current state of the art 2-QBF solvers CADET [39],CAQE [40] and DepQBF [32]. The certifying 2-QBF solver produces QBF certificates, that can be used to extract Skolem functions [8]. Developers of BaFSyn and DepQBF confirmed that the tools produce Skolem function for only valid instances, i.e. when $\forall X \exists Y F(X, Y)$ is valid. Note that the current version of CAQE does not support certification and we have used CAQE version 2 for the experiments after consultation with the developers of CAQE.

Table 1. No. of benchmarks solved by different tools

Total	BaFSyn	CAQE	DepQBF	C2Syn	BFSS	CADET	Manthan	All tools
609	13	54	59	206	247	280	**356**	476

We present the number of instances solved Table 1. Out of 609 benchmarks, the most number of instances solved by any of the remaining techniques is 280 while Manthan is able to solve 356 instances – a significant improvement over state of the art. We will focus on top 4 synthesis tools from Table 1 for further analysis.

For a deeper analysis of runtime behavior, we present the cactus plot in Fig. 2: the number of instances are shown on the x-axis and the time taken on the y-axis; a point (x, y) implies that a solver took less than or equal to y seconds to find Skolem function of x instances on a total of 609 instances. An interesting behavior predicted by cactus plot and verified upon closer analysis is that for instances that can be solved by most of the tools, the initial overhead due to a multi-phase approach may lead to relatively larger runtime for Manthan. However, with the rise in empirically observed hardness of instances, one can observe the strengths of the multi-phase approach. Overall, Manthan solves 76 more instances than the rest of the remaining techniques.

We show a pairwise comparison of Manthan vis-a-vis other techniques in Table 2. The second row of the table lists the number of instances that were solved by the technique in the corresponding column but not by Manthan while the third row lists the number of instances that were solved by Manthan but not the corresponding technique. First, we observe that Manthan solves 163, 194, and 187 instances that are not solved by C2Syn, BFSS, and CADET respectively. Though

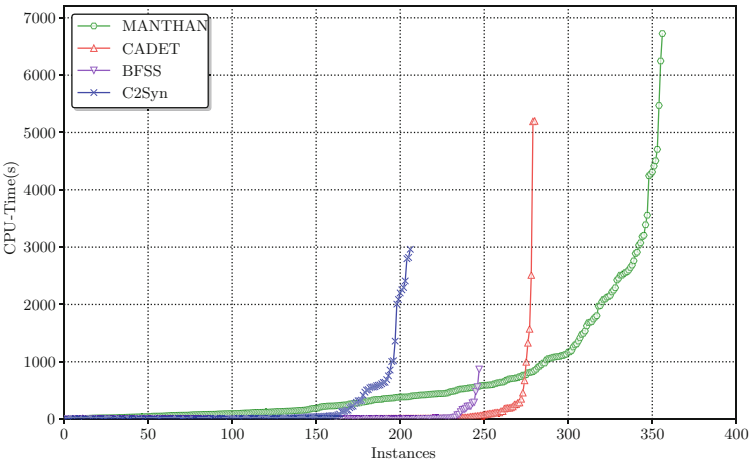


Fig. 2. Manthan versus competing tools for Skolem function synthesis

Table 2. Manthan vs other state-of-the-art tools

		C2Syn	BFSS	CADET	All Tools
Manthan	Less	13	85	111	122
	More	**163**	**194**	**187**	**60**

BFSS and CADET solve more than 80 instances that Manthan does not solve, they are not complementary; there are only 121 instances that can be solved by either BFSS or CADET but Manthan fails to solve. A closer analysis of Manthan's performance on these instances revealed that the decision trees generated by CandidateSkF were shallow, which is usually a sign of significant under-fitting. On the other hand, there are 130 instances that Manthan solves, but neither CADET nor BFSS can solve. These instances have high dependencies between variables that Manthan can infer from the samples en route to predicting good candidate Skolem functions. Akshay et al. [4] suggest that C2Syn is an orthogonal approach to BFSS. Manthan solves 81 instances that neither C2Syn nor BFSS is able to solve, and these tools together solve 86 instances that Manthan fails to solve. Overall, Manthan solves **60** instances beyond the reach of any of the above state of the art tools.

6.2 Impact of the Sampling Scheme

To analyze the impact of the adaptive sampling and the quality of distributions generated by underlying samplers, we augmented Manthan with samples drawn from different samplers for adaptive and non-adaptive sampling. In particular, we employed QuickSampler [16], KUS [42], UniGen2 [15], and BiasGen[1]. The

[1] BiasGen is developed by Mate Soos and Kuldeep S. Meel, and is pending publication.

samplers KUS and UniGen2 could only produce samples for mere 14 and 49 benchmarks respectively within a timeout of 3600 s. Hence, we have omitted KUS and UniGen2 from further analysis. We also experimented with a naive enumeration of solution using off-the-shelf SAT solver, CryptoMiniSat [43]. It is worth noting that QuickSampler performs worse than BiasGen for uniformity testing using Barbarik [13]. In our implementation, we had to turn off the validation phase of QuickSampler to allow generation number of samples within a reasonable time. To statistically validate our intuition described in Sect. 4, we performed adaptive sampling using BiasGen. We use AdaBiasGen to refer to the adaptive sampling implementation.

Table 3 presents the performance of Manthan with different samplers listed in Column 1. The columns 2, 3, and 4 lists the number of instances that were solved during the execution of respective phases: Preprocess, LearnSkF, and Refine. Finally, column 5 lists the total number of instances solved. Two important findings emerge from Table 3: Firstly, as the quality of samplers improve, so does the performance of Manthan. In particular, we observe that with the improvement in the quality of samples leads to Manthan solving more instances in LearnSkF. Secondly, we see a significant increase in the number of instances that can be solved due to LearnSkF with samples from AdaBiasGen. It is worth remarking that one should view the adaptive scheme proposed in Sect. 4 to be a proof of concept and our results will encourage the development of more complex schemes.

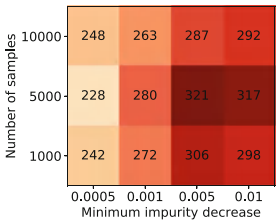


Fig. 3. Heatmap of # instances solved. (Color figure online)

Sampler	No. of instances solved			#Solved
	Preprocess	LearnSkF	Refine	
CryptoMiniSat	66	14	191	271
QuickSampler	66	28	181	275
BiasGen	66	51	228	345
AdaBiasGen	66	66	224	**356**

Table 3. Manthan with different samplers

6.3 Impact of LearnSkF

To analyze the impact of different design choices in LearnSkF, we analyzed the performance of Manthan for different samples (1000, 5000 and 10000) generated by GetSamples and for different choices of minimum impurity decrease (0.001, 0.005, 0.0005). Figure 3 shows a heatmap on the number of instances solved on each combination of the hyperparameters; the closer the color of a cell is to the red end of the spectrum, the better the performance of Manthan.

At the first look, Fig. 3 presents a puzzling picture: It seems that increasing the number of samples does not improve the performance of Manthan. On a closer analysis, we found that the increase in the number of samples leads to an increase in the runtime of CandidateSkF but without significantly increasing the number

of instances solved during LearnSkF. The runtime of CandidateSkF is dependent on the number of samples and $|Y|$. On the other hand, we see an interesting trend with respect to minimum impurity decrease where the performance first improves and then degrades. A plausible explanation for such a behavior is that with an increase in *minimum impurity decrease*, the generated decision trees tend to underfit while significantly low values of *minimum impurity decrease* lead to overfitting. We intend to study this in detail in the future.

Based on the above observations, we set the value of minimum impurity decrease to 0.005 and set the number of samples to (1) 10000 for $|Y| < 1200$, (2) 5000 for $1200 < |Y| \leq 4000$, and (3) 1000 for $|Y| > 4000$.

6.4 Division of Time Taken Across Different Phases

To analyze the time taken by different phases of Manthan across different categories of the benchmarks, we normalize the time taken for each of the four core subroutines, Preprocess, GetSamples, CandidateSkF, and RefineSkF, for every benchmark that was solved by Manthan such that the sum of time taken for each benchmark is 1. We then compute the mean of the normalized times across different categories instances. Figure 4 shows the distribution of mean normalized times for different categories: Arithmetic, Disjunction, Factorization, QBFEval, and all the instances.

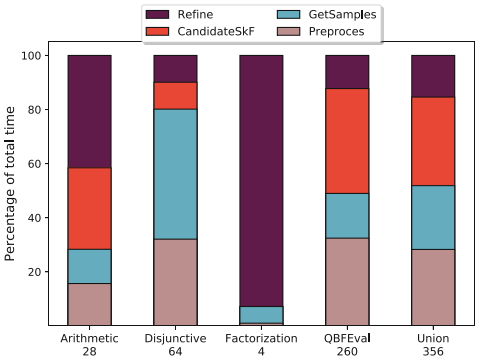


Fig. 4. Fraction of time spent in different phases in Manthan over different classes of benchmarks. (Color figure online)

The diversity of our benchmark suite shows a nuanced picture and shows that the time taken by different phases strongly depends on the family of instances. For example, the disjunctive instances are particularly hard to sample and an improvement in the sampling techniques would lead to significant performance gains. On the other hand, a significant fraction of runtime is spent in the CandidateSkF subroutine indicating the potential gains due to improvement in decision tree generation routines. In all, Fig. 4 identifies the categories

of instances that would benefit from algorithmic and engineering improvements in Manthan's different subroutines.

6.5 Impact of Using MaxSAT

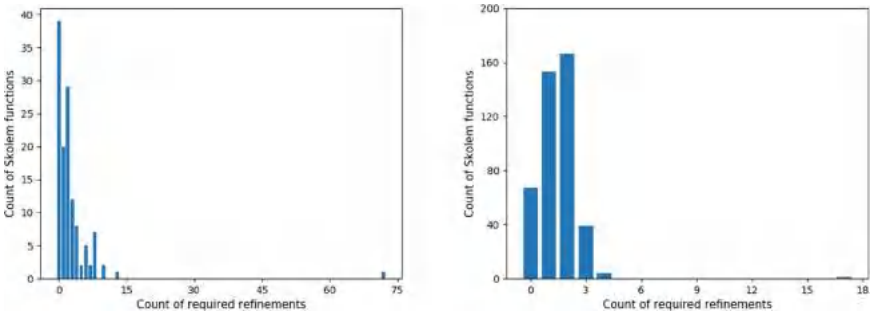
In RefineSkF, Manthan invokes the MaxSATList subroutine, which calls MaxSAT solver to identify the potential erring Skolem functions. To observe the impact of using MaxSAT solver to identify the candidates to refine, we did an experiment with Manthan, without MaxSATList subroutine call. For all y_i, where $\sigma[y_i] \neq \sigma[y_i']$ were considered as candidates to refine. Manthan without MaxSATList subroutine call solved 204 instances that represents a significant drop in the number of solved instances by Manthan with MaxSATList subroutine.

6.6 Impact of Self-substitution

To understand the impact of self-substitution, we profile the behavior of candidate Skolem functions with respect to number of refinements for two of our benchmarks; *pdtpmsmiim-all-bit* and *pdtpmsmiim*. In Fig. 5, we use histograms with the number of candidate Skolem functions on y-axis and required number of refinements on x-axis. A bar of height a i.e $y = a$ at b i.e $x = b$ in Fig. 5 represents that a candidate Skolem functions converged in b refinements. The histograms show that only a few Skolem functions require a large number of refinements: the tiny bar towards the right end in Fig. 5a represents that for the benchmark *pdtpmsmiim-all-bit* only 1 candidate Skolem function required more than 60 refinements whereas all other candidate Skolem functions needed less than 15 refinements. Similarly, for the benchmark *pdtpmsmiim*, Fig. 5b shows that only 1 candidate Skolem function was refined more than 15 times, whereas all other Skolem functions required less than 5 refinements. We found similar behaviors in many of our other benchmarks.

Based on the above trend and an examination of the decision trees corresponding to these instances, we hypothesize that some Skolem functions are hard to learn through data. For such functions, the candidate Skolem function generated from the data-driven phase in Manthan tends to be poor, and hence Manthan requires a long series of refinements for convergence. Since our refinement algorithm is designed for small, efficient corrections, we handle such hard to learn Skolem functions by synthesizing via self-substitution. Manthan detects such functions via a threshold on the number of refinements, which is empirically determined as 10, to identify hard to learn instances and sets them up for self-substitution.

In our experiments, we found 75 instances out of 356 solved instances required self-substitution, and for 51 of these 75 instances, only one variable undergoes self-substitution. Table 4 shows the impact of self-substitution for five of our benchmarks: Manthan has significant performance improvement with self-substitution in terms of the required number of refinements, which in turns affects the overall time. Note that Manthan can refine multiple candidates in a single RefineSkF call. For the first four benchmarks, all the other Skolem function



(a) Benchmark *pdtpmsmiim-all-bit:* plot for no. of Skolem functions vs required no. of refinements

(b) Benchmark *pdtpmsmiim:* plot for no. of Skolem functions vs required no. of refinements

Fig. 5. The plots to show the required number of refinements for the candidate Skolem functions.

except the poor candidates were synthesized earlier than 10 refinement iteration, and at the 10^{th} refinement iteration the poor candidate functions hit our threshold for self-substitution. Taking the case of the last benchmark, all the other Skolem functions for it were synthesized earlier than 40 refinement cycles, and the last 16 iterations were only needed for 2 of the poor candidate functions to hit our threshold for self-substitution. Note that self-substitution can lead to an exponential blowup in the size of the formula, but it works quite well in our design as most Skolem functions are learnt quite well in the LearnSkF phase.

Table 4. Manthan : Impact of self substitution

Benchmarks $\exists YF(X,Y)$	$\lvert X\rvert$	$\lvert Y\rvert$	No. of Refinements		Time(s)	
			Self-Substitution		Self-Substitution	
			Without	With	Without	With
kenflashpo2-all-bit	71	32	319	10	35.88	19.22
eijkbs1512	316	29	264	10	42.88	32.35
pdtpmsmiim-all-bit	429	30	313	10	72.75	36.08
pdtpmssfeistel	1510	68	741	10	184.11	115.07
pdtpmsmiim	418	337	127	56	1049.29	711.48

7 Conclusion

Boolean functional synthesis is a fundamental problem in Computer Science with a wide variety of applications. In this work, we propose a novel data-driven approach to synthesis that employs constrained sampling techniques for generation of data, machine learning for candidate Skolem functions, and automated reasoning to verify and refine to generate Skolem functions. Our approach achieves significant performance improvements. As pointed out in Sects. 5 and 6, our work opens up several interesting directions for future work at the intersection of machine learning, constrained sampling, and automated reasoning.

Acknowledgment. We are grateful to the anonymous reviewers and Dror Fried for constructive comments that significantly improved the final version of the paper. We are grateful to Mate Soos for tweaking BiasGen to support Manthan. We are indebted to S. Akshay, Supratik Chakraborty, and Shetal Shah for their patient responses to our tens of queries regarding prior work.

This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore: https://www.nscc.sg [1].

References

1. ASTAR, NTU, NUS, SUTD: National Supercomputing Centre (NSCC) Singapore (2018). https://www.nscc.sg/about-nscc/overview/
2. QBF solver evaluation portal 2017. http://www.qbflib.org/qbfeval17.php
3. QBF solver evaluation portal 2018. http://www.qbflib.org/qbfeval18.php
4. Akshay, S., Arora, J., Chakraborty, S., Krishna, S., Raghunathan, D., Shah, S.: Knowledge compilation for boolean functional synthesis. In: Proc. of FMCAD (2019)
5. Akshay, S., Chakraborty, S., Goel, S., Kulal, S., Shah, S.: What's hard about boolean functional synthesis? In: Proc. of CAV (2018)
6. Akshay, S., Chakraborty, S., John, A.K., Shah, S.: Towards parallel boolean functional synthesis. In: Proc. of TACAS (2017)
7. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Proc. of FMCAD (2013)
8. Balabanov, V., Jiang, J.H.R.: Resolution proofs and skolem functions in QBF evaluation and applications. In: Proc. of CAV (2011)
9. Balabanov, V., Jiang, J.H.R.: Unified QBF certification and its applications. In: Proc. of FMCAD (2012)
10. Bavishi, R., Pandey, A., Roy, S.: To be precise: regression aware debugging. In: Proc. of OOPSLA (2016)
11. Biere, A.: PicoSAT essentials. Proc. of JSAT (2008)
12. Boole, G.: The mathematical analysis of logic. Philosophical Library (1847)
13. Chakraborty, S., Meel, K.S.: On testing of uniform samplers. In: Proc. of AAAI (2019)

14. Chakraborty, S., Fried, D., Tabajara, L.M., Vardi, M.Y.: Functional synthesis via input-output separation. In: Proc. of FMCAD (2018)
15. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: Proc. of DAC (2014)
16. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: Proc. of ICSE (2018)
17. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ICE learning for synthesizing invariants and contracts. In: Proc. of OOPSLA (2018)
18. Fedyukovich, G., Gupta, A.: Functional synthesis with examples. In: Proc. of CP (2019)
19. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based boolean functional synthesis. In: Proc. of CAV (2016)
20. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Proc. of CAV (2014)
21. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proc. of POPL (2016)
22. Golia, P., Roy, S., Meel, K.S.: Manthan: A data driven approach for boolean function synthesis (2020). https://arxiv.org/abs/2005.06922
23. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: Proc. of POPL (2005)
24. Gupta, R., Sharma, S., Roy, S., Meel, K.S.: WAPS: Weighted and projected sampling. In: Proc. of TACAS (2019)
25. Heule, M.J., Seidl, M., Biere, A.: Efficient extraction of skolem functions from QRAT proofs. In: Proc. of FMCAD (2014)
26. Janota, M.: Towards generalization in QBF solving via machine learning. In: Proc. of AAAI (2018)
27. Jo, S., Matsumoto, T., Fujita, M.: SAT-based automatic rectication and debugging of combinational circuits with lut insertions. Proc. of IPSJ T-SLDM (2014)
28. John, A.K., Shah, S., Chakraborty, S., Trivedi, A., Akshay, S.: Skolem functions for factored formulas. In: Proc. of FMCAD (2015)
29. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proc. of PLDI (2011)
30. Jussila, T., Biere, A., Sinz, C., Kröning, D., Wintersteiger, C.M.: A first step towards a unified proof checker for QBF. In: Proc. of SAT (2007)
31. Logic, B., Group, V.: ABC: A system for sequential synthesis and verification. http://www.eecs.berkeley.edu/~alanmi/abc/
32. Lonsing, F., Egly, U.: Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In: Proc. of CADE (2017)
33. Löwenheim, L.: Über die auflösung von gleichungen im logischen gebietekalkul. Mathematische Annalen (1910)
34. Martins, R., Manquinho, V., Lynce, I.: Open-WBO: A modular MaxSAT solver. In: Proc. of SAT (2014)
35. Massacci, F., Marraro, L.: Logical cryptanalysis as a SAT problem. Journal of Automated Reasoning (2000)
36. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF. In: Proc. of SAT (2012)
37. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine Learning in Python. Proc. of Machine Learning Research (2011)

38. Quinlan, J.R.: Induction of decision trees. Proc. of Machine learning (1986)
39. Rabe, M.N.: Incremental determinization for quantier elimination and functional synthesis. In: Proc. of CAV (2019)
40. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Proc. of FMCAD (2015)
41. Rabe, M.N., Tentrup, L., Rasmussen, C., Seshia, S.A.: Understanding and extending incremental determinization for 2QBF. In: Proc. of CAV (2018)
42. Sharma, S., Gupta, R., Roy, S., Meel, K.S.: Knowledge compilation meets uniform sampling. In: Proc. of LPAR (2018)
43. Soos, M.: msoos/cryptominisat (2019). https://github.com/msoos/cryptominisat
44. Srivastava, S., Gulwani, S., Foster, J.S.: Template-based program verication and program synthesis. STTT (2013)
45. Tabajara, L.M., Vardi, M.Y.: Factored boolean functional synthesis. In: Proc. of FMCAD (2017)
46. Verma, S., Roy, S.: Synergistic debug-repair of heap manipulations. In: Proc. of ESEC/FSE (2017)

Decidable Synthesis of Programs with Uninterpreted Functions

Paul Krogmeier(✉), Umang Mathur, Adithya Murali, P. Madhusudan, and Mahesh Viswanathan

University of Illinois at Urbana-Champaign, Champaign, USA
{paulmk2,umathur3,adithya5,madhu,vmahesh}@illinois.edu

Abstract. We identify a decidable synthesis problem for a class of programs of unbounded size with conditionals and iteration that work over infinite data domains. The programs in our class use uninterpreted functions and relations, and abide by a restriction called coherence that was recently identified to yield decidable verification. We formulate a powerful grammar-restricted (syntax-guided) synthesis problem for coherent uninterpreted programs, and we show the problem to be decidable, identify its precise complexity, and also study several variants of the problem.

1 Introduction

Program synthesis is a thriving area of research that addresses the problem of automatically constructing a program that meets a user-given specification [1,21,22]. Synthesis specifications can be expressed in various ways: as input-output examples [19,20], temporal logic specifications for reactive programs [44], logical specifications [1,4], etc. Many targets for program synthesis exist, ranging from transition systems [31,44], logical expressions [1], imperative programs [51], distributed transition systems/programs [38,43,45], filling holes in programs [51], or repairs of programs [49].

A classical stream of program synthesis research is one that emerged from a problem proposed by Church [13] in 1960 for Boolean circuits. Seminal results by Büchi and Landweber [9] and Rabin [48] led to a mature understanding of the problem, including connections to infinite games played on finite graphs and automata over infinite trees (see [18,32]). Tractable synthesis for temporal logics like LTL, CTL, and their fragments was investigated and several applications for synthesizing hardware circuits emerged [6,7].

In recent years, the field has taken a different turn, tackling synthesis of programs that work over infinite domains such as strings [19,20], integers [1,51], and heaps [47]. Typical solutions derived in this line of research involve (a) bounding the class of programs to a finite set (perhaps iteratively increasing the class) and (b) searching the space of programs using techniques like symmetry-reduced enumeration, SAT solvers, or even random walks [1,4], typically guided

Paul Krogmeier and Mahesh Viswanathan are partially supported by NSF CCF 1901069. Umang Mathur is partially supported by a Google PhD Fellowship.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 634–657, 2020.
https://doi.org/10.1007/978-3-030-53291-8_32

by counterexamples (CEGIS) [28,34,51]. Note that iteratively searching larger classes of programs allows synthesis engines to find a program if one exists, but it does not allow one to conclude that there is no program that satisfies the specification. Consequently, in this stream of research, decidability results are uncommon (see Sect. 7 for some exceptions in certain heavily restricted cases).

In this paper we present, to the best of our knowledge, the first decidability results for program synthesis over a natural class of programs with iteration/recursion, having arbitrary sizes, and which work on infinite data domains. In particular, we show decidable synthesis of a subclass of programs that use uninterpreted functions and relations.

Our primary contribution is a decidability result for realizability and synthesis of a restricted class of imperative *uninterpreted* programs. Uninterpreted programs work over infinite data models that give arbitrary meanings to their functions and relations. Such programs satisfy their assertions if they hold along all executions for *every* model that interprets the functions and relations. The theory of uninterpreted functions and relations is well studied—classically, in 1929, by Gödel, where completeness results were shown [5] and, more recently, its decidable quantifier-free fragment has been exploited in SMT solvers in combination with other theories [8]. In recent work [39], a subclass of uninterpreted programs, called *coherent* programs, was identified and shown to have a decidable verification problem. Note that in this verification problem there are no user-given loop invariants; the verification algorithm finds inductive invariants and proves them automatically in order to prove program correctness.

In this paper, we consider the synthesis problem for coherent uninterpreted programs. The user gives a *grammar* $\mathcal{G}$ that generates well-formed programs in our programming language. The grammar can force programs to have **assert** statements at various points which collectively act as the specification. The program synthesis problem is then to construct a coherent program, if one exists, conforming to the grammar $\mathcal{G}$ that satisfies all assertions in all executions when running on *any* data model that gives meaning to function and relation symbols.

Our primary result is that the realizability problem (checking the existence of a program conforming to the grammar and satisfying its assertions) is decidable for coherent uninterpreted programs. We prove that the problem is 2EXPTIME-complete. Further, whenever a correct coherent program that conforms to the grammar exists, we can synthesize one. We also show that the realizability/synthesis problem is undecidable if the coherence restriction is dropped. In fact we show a stronger result that the problem is undecidable even for synthesis of *straight-line* programs (without conditionals and iteration)!

Coherence of programs is a technical restriction that was introduced in [39]. It consists of two properties, both of which were individually proven to be essential for ensuring that program verification is decidable. Intuitively, the restriction demands that functions are computed on any tuple of terms only once and that assumptions of equality come early in the executions. In more recent work [41], the authors extend this decidability result to handle map updates, and applied it to memory safety verification for a class of heap-manipulating programs on

forest data-structures, demonstrating that the restriction of coherence is met in practice by certain natural and useful classes of programs.

Note that automatic synthesis of correct programs over infinite domains demands that we, at the very least, can automatically verify the synthesized program to be correct. The class of coherent uninterpreted programs identified in the work of [39] is the only natural class of programs we are aware of that has recursion and conditionals, works over infinite domains, and admits decidable verification. Consequently, this class is a natural target for proving a decidable synthesis result.

The problem of synthesizing a program from a grammar with assertions is a powerful formulation of program synthesis. In particular, the grammar can be used to restrict the space of programs in various ways. For example, we can restrict the space syntactically by disallowing while loops. Or, for a fixed n, by using a set of Boolean variables linear in n and requiring a loop body to strictly increment a counter encoded using these variables, we can demand that loops terminate in a linear/polynomial/exponential number of iterations. We can also implement loops that do not always terminate, but terminate only when the data model satisfies a particular property, e.g., programs that terminate only on finite list segments, by using a skeleton of the form: **while** $(\mathtt{x} \neq \mathtt{y})\{ \ldots ; \ \mathtt{x} := \mathtt{next(x)}\}$. Grammar-restricted program synthesis can express the synthesis of programs with holes, used in systems like SKETCH [50], where the problem is to fill holes using programs/expressions conforming to a particular grammar so that the assertions in the program hold. Synthesizing programs or expressions using restricted grammars is also the cornerstone of the intensively studied SYGUS (syntax-guided synthesis) format [1,52][1].

The proof of our decidability result relies on tree automata, a callback to classical theoretical approaches to synthesis. The key idea is to represent programs as trees and build automata that accept trees corresponding to correct programs. The central construction is to build a two-way alternating tree automaton that accepts *all* program trees of coherent programs that satisfy their assertions. Given a grammar $\mathcal{G}$ of programs (which has to satisfy certain natural conditions), we show that there is a regular set of program trees for the language of allowed programs $L(\mathcal{G})$. Intersecting the automata for these two regular tree languages and checking for emptiness establishes the upper bound. Our constructions crucially use the automaton for verifying coherent uninterpreted programs in [39] and adapt ideas from [35] for building two-way automata over program trees. Our final decision procedure is doubly-exponential in the number of program variables and *linear* in the size of the grammar. We also prove a matching lower bound by reduction from the acceptance problem for alternating exponential-space Turing machines. The reduction is non-trivial in that programs (which correspond to runs in the Turing machine) must simulate sequences of configurations, each of which is of exponential size, by using only polynomially-many variables.

[1] Note, however, that both SKETCH and SYGUS problems are defined using functions and relations that are interpreted using standard theories like arithmetic, etc., and hence of course do not have decidable synthesis.

Recursive Programs, Transition Systems, and Boolean Programs: We study three related synthesis problems. First, we show that our results extend to synthesis of call-by-value *recursive* uninterpreted programs (with a fixed number of functions and fixed number of local/global variables). This problem is also 2EXPTIME-complete but is more complex, as even single executions simulated on the program tree must be split into separate copies, with one copy executing the summary of a function call and the other proceeding under the assumption that the call has returned in a summarized state.

We next examine a synthesis problem for *transition systems*. Transition systems are similar to programs in that they execute similar kinds of atomic statements. We allow the user to restrict the set of allowable executions (using regular sets). Despite the fact that this problem seems very similar to program synthesis, we show that it is an *easier* problem, and coherent transition system realizability and synthesis can be solved in time exponential in the number of program variables and polynomial in the size of the automata that restrict executions. We prove a corresponding lower bound to establish EXPTIME-completeness of this problem.

Finally, we note that our results also show, as a corollary, that the grammar-restricted realizability/synthesis problem for Boolean programs (resp. execution-restricted synthesis problem for Boolean transition systems) is decidable and is 2EXPTIME-complete (resp. EXPTIME-complete). These results for Boolean programs are themselves new. The lower bound results for these problems hence show that coherent program/transition-system synthesis is not particularly harder than Boolean program synthesis for uninterpreted programs. Grammar-restricted Boolean program synthesis is an important problem which is addressed by many practical synthesis systems like Sketch [50].

Due to space restrictions, we present only proof gists for main results in the paper. All the complete proofs can be found in our technical report [30].

2 Examples

We will begin by looking at several examples to gain some intuition for uninterpreted programs.

Example 1. Consider the program in Fig. 1 (left). This program has a *hole* '$\langle\langle$ `??` | Cannot ... $\rangle\rangle$' that we intend to fill with a sub-program so that the entire program (together with the contents of the hole) satisfies the assertion at the end. The sub-program corresponding to the hole is allowed to use the variable `cipher` as well as some additional variables $\mathtt{y}_1, \ldots, \mathtt{y}_n$ (for some fixed n), but is not allowed to refer to `key` or `secret` in any way. Here we also restrict the hole to exclude while loops. This example models the encryption of a secret message `secret` with a key `key`. The assumption in the second line of the program models

```
cipher := enc(secret, key);
assume(secret = dec(cipher, key));
⟨⟨ ?? | Cannot refer to secret or key ⟩⟩;
assert(z = secret)
```

Decrypting a ciphertext

```
assume(T ≠ F);
if (x = T) then b := T else b := F;
⟨⟨ ?? | Cannot refer to x or b ⟩⟩;
assert(y = b)
```

Synthesis with incomplete information

Fig. 1. Examples of programs with holes

the fact that the secret message can be decrypted from `cipher` and `key`. Here, the functions `enc` and `dec` are *uninterpreted functions*, and thus the program we are looking for is an *uninterpreted program*. For such a program, the assertion "**assert**(`z = secret`)" holds at the end if it holds for *all models*, i.e, for all interpretations of `enc` and `dec` and for all initial values of the program variables `secret`, `key`, `cipher`, and $\mathtt{y}_1, \ldots, \mathtt{y}_n$. With this setup, we are essentially asking whether a program that does not have access to `key` can recover `secret`. It is not hard to see that there is no program which satisfies the above requirement. The above modeling of keys, encryption, nonces, etc. is common in algebraic approaches to modeling cryptographic protocols [15,16].

Example 2. The program in Fig. 1 (right) is another simple example of an unrealizable specification. The program variables here are `x`, `b`, and `y`. The hole in this partial program is restricted so that it cannot refer to `x` or `b`. It is easy to phrase the question for synthesis of the complete program in terms of a grammar. The restriction on the hole ensures that the synthesized code fragment can neither directly check if `x = T`, nor indirectly check via `b`. Consequently, it is easy to see that there is no program for the hole that can ensure `y` is equal to `b`. We remark that the code at the hole, apart from not being allowed to examine some variables, is also implicitly prohibited from looking at the control path taken to reach the hole. If we could synthesize two different programs depending on the control path taken to reach the hole, then we could set `y := T` when the **then**-branch is taken and set `y := F` when the **else**-branch is taken. Program synthesis requires a control-flow independent decision to be made about how to fill the hole. In this sense, we can think of the hole as having only *incomplete information* about the executions for which it must be correct. This can be used to encode specifications using complex ghost code, as we show in the next examples. In Sect. 6, we explore a slightly different synthesis problem, called *transition system synthesis*, where holes can be differently instantiated based on the history of an execution.

Example 3. In this example, we model the synthesis of a program that checks whether a linked list pointed to by some node `x` has a key `k`. We model a *next* pointer with a unary function `next` and we model locations using elements in the underlying data domain.

Our formalism allows only for **assert** statements to specify desired program properties. In order to state the correctness specification for our desired

list-search program, we interleave *ghost code* into the program skeleton; we distinguish ghost code fragments by enclosing them in dashed boxes. The skeleton in Fig. 2 has a loop that advances the pointer variable `x` along the list until `NIL` is reached. We model `NIL` with an immutable program variable. The first hole '$\langle\langle$??^{①} $\rangle\rangle$' before the **while**-loop and the second hole '$\langle\langle$??^{②} $\rangle\rangle$' within the **while**-loop need to be filled so that the assertion at the end is satisfied. We use three ghost variables in the skeleton: g_{ans}, $g_{witness}$, and g_{found}. The ghost variable g_{ans} evaluates to whether we expect to find `k` in the list or not, and hence at the end the skeleton asserts that the Boolean variable `b` computed by the holes is precisely g_{ans}. The holes are restricted to not look at the ghost variables.

Now, notice that the skeleton needs to *check* that the answer g_{ans} is indeed correct. If g_{ans} is not `T`, then we add the assumption that `key(x)` $\neq$ `k` in each iteration of the loop, hence ensuring the key is not present. For ensuring correctness in the case $g_{ans} = T$, we need two more ghost variables $g_{witness}$ and g_{found}. The variable $g_{witness}$ witnesses the precise location in the list that holds the key `k`, and variable g_{found} indicates whether the location at $g_{witness}$ belongs to the list pointed to by `x`. Observe that this specification can be realized by filling '$\langle\langle$??^{①} $\rangle\rangle$' with "`b` := `F`" and '$\langle\langle$??^{②} $\rangle\rangle$' with "**if** `key(x)` = `k` **then** `b` := `T`", for instance. Furthermore, this program is *coherent* [39] and hence our decision procedure will answer in the affirmative and synthesize code for the holes.

```
assume(T ≠ F);
g_found := F;
⟨⟨ ??① ⟩⟩;
while(x ≠ NIL) {
    if (g_ans ≠ T) then
        assume(key(x) ≠ k);
    else if (g_witness = x) then {
        assume (key(x) = k);
        g_found := T;
    };
    ⟨⟨ ??② ⟩⟩;
    x := next(x);
}
assume (g_ans = T ⇒ g_found = T);
assert b = T ⟺ g_ans = T
```

Fig. 2. Skeleton with ghost code

In fact, our procedure will synthesize a representation for *all* possible ways to fill the holes (thus including the solution above) and it is therefore possible to enumerate and pick specific solutions. It is straightforward to formulate a grammar which matches this setup. As noted, we must stipulate that the holes do not use the ghost variables.

Example 4. Consider the same program skeleton as in Example 3, but let us add an assertion at the end: "**assert** (`b` = `T` $\Rightarrow$ `z` = $g_{witness}$)", where `z` is another program variable. We are now demanding that the synthesized code also find a location `z`, whose key is `k`, that is equal to the ghost location $g_{witness}$, which is guessed nondeterministically at the beginning of the program. This specification is *unrealizable*: for a list with multiple locations having the key `k`, no matter what the program picks we can always take $g_{witness}$ to be the *other* location with key `k` in the list, thus violating the assertion. Our decision procedure will report in the negative for this specification.

Example 5 (Input/Output Examples). We can encode input/output examples by adding a sequence of assignments and assumptions that define certain models at

the beginning of the program grammar. For instance, the sequence of statements in Fig. 3 defines a linked list of two elements with different keys.

We can similarly use special variables to define the output that we expect in the case of each model. And as we saw in the ghost code of Fig. 2, we can use fresh variables to introduce nondeterministic choices, which the grammar can use to pick an example model nondeterministically. Thus when the synthesized program is executed on the chosen model it computes the expected answer. This has the effect of requiring a solution that generalizes across models. See [30] for a more detailed example.

```
assume(x1 ≠ NIL);
x2 := next(x1);
assume(x2 ≠ NIL);
assume(next(x2) = NIL);
k1 := key(x1);
k2 := key(x2);
assume(k1 ≠ k2)
```

Fig. 3. An example model

3 Preliminaries

In this section we define the syntax and semantics of uninterpreted programs and the *(grammar-restricted) uninterpreted program synthesis* problem.

Syntax. We fix a first order signature $\Sigma = (\mathcal{F}, \mathcal{R})$, where $\mathcal{F}$ and $\mathcal{R}$ are sets of function and relation symbols, respectively. Let V be a finite set of program variables. The set of programs over V is inductively defined using the following grammar, with $f \in \mathcal{F}$, $R \in \mathcal{R}$ (with f and R of the appropriate arities), and $x, y, z_1, \ldots, z_r \in V$.

$$\begin{aligned}\langle stmt\rangle_V ::= \;& \textbf{skip} \mid x := y \mid x := f(z_1, \ldots, z_r) \mid \\ & \textbf{assume}\left(\langle cond\rangle_V\right) \mid \textbf{assert}\left(\langle cond\rangle_V\right) \mid \langle stmt\rangle_V \,;\, \langle stmt\rangle_V \mid \\ & \textbf{if}\left(\langle cond\rangle_V\right) \textbf{then}\, \langle stmt\rangle_V \,\textbf{else}\, \langle stmt\rangle_V \mid \textbf{while}\left(\langle cond\rangle_V\right) \langle stmt\rangle_V \\ \langle cond\rangle_V ::= \;& x = y \mid R(z_1, \ldots, z_r) \mid \langle cond\rangle_V \vee \langle cond\rangle_V \mid \neg\langle cond\rangle_V\end{aligned}$$

Without loss of generality, we can assume that our programs do not use relations (they can be modeled with functions) and that every condition is either an equality or disequality between variables (arbitrary Boolean combinations can be modeled with nested **if**−**then**−**else**). When the set of variables V is clear from context, we will omit the subscript V from $\langle stmt\rangle_V$ and $\langle cond\rangle_V$.

Program Executions. An execution over V is a finite word over the alphabet

$$\begin{aligned}\Pi_V = \{&\text{“}x := y\text{”}, \text{“}x := f(\overline{z})\text{”}, \text{“}\textbf{assume}(x = y)\text{”}, \text{“}\textbf{assume}(x \neq y)\text{”}, \\ &\text{“}\textbf{assert}(\bot)\text{”} \mid x, y \in V, \overline{z} \in V^r, f \in \mathcal{F}\}.\end{aligned}$$

The set of *complete executions* for a program p over V, denoted $\mathsf{Exec}(p)$, is a regular language. See [30] for a straightforward definition. The set $\mathsf{PExec}(p)$ of *partial executions* is the set of prefixes of complete executions in $\mathsf{Exec}(p)$. We refer to partial executions as simply *executions*, and clarify as needed when the distinction is important.

Semantics. The semantics of executions is given in terms of data models. A data model $\mathcal{M} = (U, \mathcal{I})$ is a first order structure over Σ comprised of a universe U and an interpretation function $\mathcal{I}$ for the program symbols. The semantics of an execution π over a data model $\mathcal{M}$ is given by a configuration $\sigma(\pi, \mathcal{M}) : V \to U$ which maps each variable to its value in the universe U at the end of π. This notion is straightforward and we skip the formal definition (see [39] for details). For a fixed program p, any particular data model corresponds to at most one complete execution $\pi \in \mathsf{Exec}(p)$.

An execution π is *feasible* in a data model $\mathcal{M}$ if for every prefix $\rho = \rho' \cdot \mathbf{assume}(x \sim y)$ of π (where $\sim \in \{=, \neq\}$), we have $\sigma(\rho', \mathcal{M})(x) \sim \sigma(\rho', \mathcal{M})(y)$. Execution π is said to be *correct* in a data model $\mathcal{M}$ if for every prefix of π of the form $\rho = \rho' \cdot \mathbf{assert}(\bot)$, we have that ρ' is not feasible, or *infeasible* in $\mathcal{M}$. Finally, a program p is said to be *correct* if for all data models $\mathcal{M}$ and executions $\pi \in \mathsf{PExec}(p)$, π is correct in $\mathcal{M}$.

3.1 The Program Synthesis Problem

We are now ready to define the program synthesis problem. Our approach will be to allow users to specify a grammar and ask for the synthesis of a program from the grammar. We allow the user to express specifications using *assertions* in the program to be synthesized.

Grammar Schema and Input Grammar. In our problem formulation, we allow users to define a grammar which conforms to a schema, given below. The input grammars allow the usual context-free power required to describe proper nesting/bracketing of program expressions, but disallow other uses of the context-free power, such as *counting statements*.

For example, we disallow the grammar in Fig. 4. This grammar has two non-terminals S (the start symbol) and T. It generates programs with a conditional that has the *same* number of assignments in the **if** and **else** branches. We assume a countably infinite set PN of nonterminals and a countably infinite set PV of program variables. The grammar schema $\mathcal{S}$ over PN and PV is an infinite collection of productions:

$$S \to \mathbf{if}\ (\mathtt{x} = \mathtt{y})\ \mathbf{then}\ \mathtt{u} := \mathtt{v}\ T\ \mathtt{u} := \mathtt{v}$$
$$T \to \mathbf{else}$$
$$T \to ;\ \mathtt{u} := \mathtt{v}\ \ T\ \ \mathtt{u} := \mathtt{v}\ ;$$

Fig. 4. Grammar with counting

$$\mathcal{S} = \left\{ \begin{array}{l} \text{“}P \to x := y\text{”}, \text{“}P \to x := f(\overline{z})\text{”}, \\ \text{“}P \to \mathbf{assume}(x \sim y)\text{”}, \text{“}P \to \mathbf{assert}(\bot)\text{”}, \\ \text{“}P \to \mathbf{skip}\text{”}, \text{“}P \to \mathbf{while}\,(x \sim y)\ P_1\text{”}, \\ \text{“}P \to \mathbf{if}\,(x \sim y)\,\mathbf{then}\,P_1\,\mathbf{else}\,P_2\text{”}, \text{“}P \to P_1; P_2\text{”} \end{array} \middle| \begin{array}{l} P, P_1, P_2 \in PN \\ x, y \in PV, \overline{z} \in PV^r \\ \sim \in \{=, \neq\} \end{array} \right\}$$

An *input grammar* $\mathcal{G}$ is any finite subset of the schema $\mathcal{S}$, and it defines a set of programs, denoted $L(\mathcal{G})$. We can now define the main problem addressed in this work.

Definition 1 (Uninterpreted Program Realizability and Synthesis). *Given an input grammar $\mathcal{G}$, the realizability problem is to determine whether there is an uninterpreted program $p \in L(\mathcal{G})$ such that p is correct. The synthesis problem is to determine the above, and further, if realizable, synthesize a correct program $p \in L(\mathcal{G})$.*

Example 6. Consider the program with a hole from Example 1 (Fig. 1, left). We can model that synthesis problem in our framework with the following grammar.

$S \to P_1; P_2; P_{\langle\langle \texttt{??} \rangle\rangle}; P_3$
$P_1 \to$ "`cipher := enc(secret, key)`"
$P_2 \to$ "**assume**(`secret = dec(cipher, key)`)"
$P_{\langle\langle \texttt{??} \rangle\rangle} \to \langle stmt \rangle_{V_{\langle\langle \texttt{??} \rangle\rangle}}$
$P_3 \to$ "**assert**(`z = secret`)"

Here, $V_{\langle\langle \texttt{??} \rangle\rangle} = \{\texttt{cipher}, \texttt{y}_1, \ldots, \texttt{y}_n\}$ and the grammar $\langle stmt \rangle_{V_{\langle\langle \texttt{??} \rangle\rangle}}$ is that of Sect. 3, restricted to loop-free programs. Any program generated from this grammar indeed matches the template from Fig. 1 (left) and any such program is correct if it satisfies the last assertion for all models, i.e., all interpretations of the function symbols `enc` and `dec` and for all initial values of the variables in $V = V_{\langle\langle \texttt{??} \rangle\rangle} \cup \{\texttt{key}, \texttt{secret}\}$.

4 Undecidability of Uninterpreted Program Synthesis

Since verification of uninterpreted programs with loops is undecidable [39,42], the following is immediate.

Theorem 1. *The uninterpreted program synthesis problem is undecidable.*

We next consider synthesizing loop-free uninterpreted programs (for which verification reduces to satisfiability of quantifier-free EUF) from grammars conforming to the following schema:

$$\mathcal{S}_{\text{loop-free}} = \mathcal{S} \setminus \{\text{“}P \to \textbf{while}\,(x \sim y)\ P_1\text{”} \mid P, P_1 \in PN,\ x, y \in PV, \sim \in \{=, \neq\}\}$$

Theorem 2. *The uninterpreted program synthesis problem is undecidable for the schema $\mathcal{S}_{\textit{loop-free}}$.*

This is a corollary of the following stronger result: synthesis of *straight-line uninterpreted programs* (conforming to schema $\mathcal{S}_{\mathsf{SLP}}$ below) is undecidable.

$$\mathcal{S}_{\mathsf{SLP}} = \mathcal{S}_{\text{loop-free}} \setminus \{\text{“}P \to \textbf{if}(x \sim y)\,\textbf{then}\,P_1\,\textbf{else}\,P_2\text{”} \mid P, P_1, P_2 \in PN, \\ x, y \in PV, \sim \in \{=, \neq\}\}$$

Theorem 3. *The uninterpreted program synthesis problem is undecidable for the schema $\mathcal{S}_{\mathsf{SLP}}$.*

In summary, program synthesis of even straight-line uninterpreted programs, which have neither conditionals nor iteration, is already undecidable. The notion of *coherence* for uninterpreted programs was shown to yield decidable verification in [39]. As we'll see in Sect. 5, restricting to coherent programs yields decidable synthesis, even for programs with conditionals *and* iteration.

5 Synthesis of Coherent Uninterpreted Programs

In this section, we present the main result of the paper: grammar-restricted program synthesis for uninterpreted *coherent* programs [39] is decidable. Intuitively, coherence allows us to maintain congruence closure in a streaming fashion when reading a coherent execution. First we recall the definition of coherent executions and programs in Sect. 5.1 and also the algorithm for verification of such programs. Then we introduce the synthesis procedure, which works by constructing a two-way alternating tree automaton. We briefly discuss this class of tree automata in Sect. 5.2 and recall some standard results. In Sects. 5.3, 5.4 and 5.5 we describe the details of the synthesis procedure, argue its correctness, and discuss its complexity. In Sect. 5.6, we present a tight lower bound result.

5.1 Coherent Executions and Programs

The notion of coherence for an execution π is defined with respect to the *terms* it computes. Intuitively, at the beginning of an execution, each variable $x \in V$ stores some constant term $\widehat{x} \in \mathcal{C}$. As the execution proceeds, new terms are computed and stored in variables. Let Terms_Σ be the set of all ground terms defined using the constants and functions in Σ. Formally, the term corresponding to a variable $x \in V$ at the end of an execution $\pi \in \Pi_V^*$, denoted $\mathsf{T}(\pi, x) \in \mathsf{Terms}_\Sigma$, is inductively defined as follows. We assume that the set of constants $\mathcal{C}$ includes a designated set of *initial* constants $\widehat{V} = \{\widehat{x} \mid x \in V\} \subseteq \mathcal{C}$.

$$\begin{aligned}
\mathsf{T}(\varepsilon, x) &= \widehat{x} && x \in V \\
\mathsf{T}(\pi \cdot \text{“}x := y\text{”}, x) &= \mathsf{T}(\pi, y) && x, y \in V \\
\mathsf{T}(\pi \cdot \text{“}x := f(z_1, \ldots, z_r)\text{”}, x) &= f(\mathsf{T}(\pi, z_1), \ldots, \mathsf{T}(\pi, z_r)) && x, z_1, \ldots, z_r \in V \\
\mathsf{T}(\pi \cdot a, x) &= \mathsf{T}(\pi, x) && \text{otherwise}
\end{aligned}$$

We will use $\mathsf{T}(\pi)$ to denote the set $\{\mathsf{T}(\pi', x) \mid x \in V, \pi' \text{ is a prefix of } \pi\}$.

A related notion is the set of *term equality assumptions* that an execution accumulates, which we formalize as $\alpha : \pi \to \mathcal{P}(\mathsf{Terms}_\Sigma \times \mathsf{Terms}_\Sigma)$, and define inductively as $\alpha(\varepsilon) = \varnothing$, $\alpha(\pi \cdot \text{“}\textbf{assume}(x = y)\text{”}) = \alpha(\pi) \cup \{(\mathsf{T}(\pi, x), \mathsf{T}(\pi, y))\}$, and $\alpha(\pi \cdot a) = \alpha(\pi)$ otherwise.

For a set of term equalities $A \subseteq \mathsf{Terms}_\Sigma \times \mathsf{Terms}_\Sigma$, and two ground terms $t_1, t_2 \in \mathsf{Terms}_\Sigma$, we say t_1 and t_2 are *equivalent modulo* A, denoted $t_1 \cong_A t_2$, if $A \models t_1 = t_2$. For a set of terms $S \subseteq \mathsf{Terms}_\Sigma$, and a term $t \in \mathsf{Terms}_\Sigma$ we write $t \in_A S$ if there is a term $t' \in S$ such that $t \cong_A t'$. For terms $t, s \in \mathsf{Terms}_\Sigma$, we say s is a *superterm modulo* A of t, denoted $t \preccurlyeq_A s$ if there are terms $t', s' \in \mathsf{Terms}_\Sigma$ such that $t \cong_A t'$, $s \cong_A s'$ and s' is a superterm of t'.

With the above notation in mind, we now review the notion of coherence.

Definition 2 (Coherent Executions and Programs [39]). *An execution $\pi \in \Pi_V^*$ is said to be* coherent *if it satisfies the following two conditions.*

Memoizing. *Let $\rho = \rho' \cdot \text{“}x := f(\overline{y})\text{”}$ be a prefix of π. If $t_x = \mathsf{T}(\rho, x) \in_{\alpha(\rho')} \mathsf{T}(\rho')$, then there is a variable $z \in V$ such that $t_x \cong_{\alpha(\rho')} t_z$, where $t_z = \mathsf{T}(\rho', z)$.*

Early Assumes. *Let $\rho = \rho' \cdot$ "**assume**$(x = y)$" be a prefix of π, $t_x = \mathsf{T}(\rho', x)$ and $t_y = \mathsf{T}(\rho', y)$. If there is a term $s \in \mathsf{T}(\rho')$ such that either $t_x \preccurlyeq_{\alpha(\rho')} s$ or $t_y \preccurlyeq_{\alpha(\rho')} s$, then there is a variable $z \in V$ such that $s \cong_{\alpha(\rho')} t_z$, where $t_z = \mathsf{T}(\rho', z)$.*

A program p is coherent if every complete execution $\pi \in \mathsf{Exec}(p)$ is coherent.

The following theorems due to [39] establish the decidability of verifying coherent programs and also of checking if a program is coherent.

Theorem 4 ([39]). *The verification problem for coherent programs, i.e. checking if a given uninterpreted coherent program is correct, is decidable.*

Theorem 5 ([39]). *The problem of checking coherence, i.e. checking if a given uninterpreted program is coherent, is decidable.*

The techniques used in [39] are automata theoretic. They allow us to construct an automaton $\mathcal{A}^{\cdots}_{\mathsf{exec}}$[2], of size $O(2^{\mathrm{poly}(|V|)})$, which accepts all coherent executions that are also correct.

To give some intuition for the notion of coherence, we illustrate simple example programs that are not coherent. Consider program p_0 below, which is not coherent because it fails to be memoizing.

$$p_0 \quad \triangleq \quad \texttt{x := f(y); x := f(x); z := f(y)}$$

The first and third statements compute $f(\widehat{y})$, storing it in variables x and z, respectively, but the term is *dropped* after the second statement and hence is not contained in any program variable when the third statement executes. Next consider program p_1, which is not coherent because it fails to have early assumes.

$$p_1 \quad \triangleq \quad \texttt{x := f(w); x := f(x); y := f(z); y := f(y);} \ \textbf{assume}\texttt{(w = z)}$$

Indeed, the assume statement is not early because superterms of w and z, namely $f(\widehat{w})$ and $f(\widehat{z})$, were computed and subsequently dropped before the assume.

Intuitively, the coherence conditions are necessary to allow equality information to be tracked with finite memory. We can make this stark by tweaking the example for p_1 above as follows.

$$\begin{aligned} p_1' \quad \triangleq \quad & \texttt{x := f(w);} \underbrace{\texttt{x := f(x)} \cdots \texttt{x := f(x)}}_{n \text{ times}}\texttt{;} \\ & \texttt{y := f(z);} \underbrace{\texttt{y := f(y)} \cdots \texttt{y := f(y)}}_{n \text{ times}}\texttt{;} \ \textbf{assume}\texttt{(w = z)} \end{aligned}$$

Observe that, for large n (e.g. $n > 100$), many terms are computed and dropped by this program, like $f^{42}(\widehat{x})$ and $f^{99}(\widehat{y})$ for instance. The difficulty with this

[2] We use superscripts '$\cdots$' and '$\widehat{\ }$' for word and tree automata, respectively.

program, from a verification perspective, is that the assume statement entails equalities between many terms which have not been kept track of. Imagine trying to verify the following program

$$p_2 \quad \triangleq \quad p_1'; \texttt{assert(x = y)}$$

Let $\pi_{p_1'} \in \mathsf{Exec}(p_1')$ be the unique complete execution of p_1'. If we examine the details, we see that $t_x = \mathsf{T}(\pi_{p_1'}, x) = f^{101}(\widehat{w})$ and $t_y = \mathsf{T}(\pi_{p_1'}, y) = f^{101}(\widehat{z})$. The assertion indeed holds because $t_x \cong_{\{(\widehat{w},\widehat{z})\}} t_y$. However, to keep track of this fact requires remembering an arbitrary number of terms that grows with the size of the program. Finally, we note that the coherence restriction is met by many single-pass algorithms, e.g. searching and manipulation of lists and trees.

5.2 Overview of the Synthesis Procedure

Our synthesis procedure uses tree automata. We consider tree representations of programs, or *program trees*. The synthesis problem is thus to check if there is a program tree whose corresponding program is coherent, correct, and belongs to the input grammar $\mathcal{G}$.

The synthesis procedure works as follows. We first construct a top-down tree automaton $\mathcal{A}^{\Lambda}_{\mathcal{G}}$ that accepts the set of trees corresponding to the programs generated by $\mathcal{G}$. We next construct another tree automaton $\mathcal{A}^{\Lambda}_{\mathsf{cc}}$, which accepts all trees corresponding to programs that are coherent and correct. $\mathcal{A}^{\Lambda}_{\mathsf{cc}}$ is a two-way alternating tree automaton that simulates all executions of an input program tree and checks that each is both correct and coherent. In order to simulate longer and longer executions arising from constructs like **while**-loops, the automaton traverses the input tree and performs multiple passes over subtrees, visiting the internal nodes of the tree many times. We then translate the two-way alternating tree automaton to an equivalent (one-way) nondeterministic top-down tree automaton by adapting results from [33,53] to our setting. Finally, we check emptiness of the intersection between this top-down automaton and the grammar automaton $\mathcal{A}^{\Lambda}_{\mathcal{G}}$. The definitions for trees and the relevant automata are standard, and we refer the reader to [14] and to our technical report [30].

5.3 Tree Automaton for Program Trees

Every program can be represented as a tree whose leaves are labeled with basic statements like "$x := y$" and whose internal nodes are labeled with constructs like **while** and **seq** (an alias for the sequencing construct '**;**'), which have subprograms as children. Essentially, we represent the set of programs generated by an input grammar $\mathcal{G}$ as a regular set of program trees, accepted by a nondeterministic top-down tree automaton $\mathcal{A}^{\Lambda}_{\mathcal{G}}$. The construction of $\mathcal{A}^{\Lambda}_{\mathcal{G}}$ mimics the standard construction for tree automata that accept *parse trees* of context free grammars. The formalization of this intuition is straightforward, and we refer the reader to [30] for details. We note the following fact regarding the construction of the acceptor of program trees from a particular grammar $\mathcal{G}$.

Lemma 1. *$\mathcal{A}^{\Lambda}_{\mathcal{G}}$ has size $O(|\mathcal{G}|)$ and can be constructed in time $O(|\mathcal{G}|)$.* □

5.4 Tree Automaton for Simulating Executions

We now discuss the construction of the two-way alternating tree automaton $\mathcal{A}^{\circlearrowleft}_{\mathsf{cc}}$ that underlies our synthesis procedure. A two-way alternating tree automaton consists of a finite set of states and a transition function that maps tuples (q, m, a) of state, incoming direction, and node labels to positive Boolean formulas over pairs (q', m') of next state and next direction. In the case of our binary program trees, incoming directions come from $\{D, U_L, U_R\}$, corresponding to coming down from a parent, and up from left and right children. Next directions come from $\{U, L, R\}$, corresponding to going up to a parent, and down to left and right children.

The automaton $\mathcal{A}^{\circlearrowleft}_{\mathsf{cc}}$ is designed to accept the set of all program trees that correspond to correct and coherent programs. This is achieved by ensuring that a program tree is accepted precisely when all executions of the program it represents are accepted by the word automaton $\mathcal{A}^{\dashrightarrow}_{\mathsf{exec}}$ (Sect. 5.1). The basic idea behind $\mathcal{A}^{\circlearrowleft}_{\mathsf{cc}}$ is as follows. Given a program tree T as input, $\mathcal{A}^{\circlearrowleft}_{\mathsf{cc}}$ traverses T and explores all the executions of the associated program. For each execution σ, $\mathcal{A}^{\circlearrowleft}_{\mathsf{cc}}$ keeps track of the state that the word automaton $\mathcal{A}^{\dashrightarrow}_{\mathsf{exec}}$ would reach after reading σ. Intuitively, an accepting run of $\mathcal{A}^{\circlearrowleft}_{\mathsf{cc}}$ is one which never visits the unique rejecting state of $\mathcal{A}^{\dashrightarrow}_{\mathsf{exec}}$ during simulation.

We now give the formal description of $\mathcal{A}^{\circlearrowleft}_{\mathsf{cc}} = (Q^{\mathsf{cc}}, I^{\mathsf{cc}}, \delta^{\mathsf{cc}}_0, \delta^{\mathsf{cc}}_1, \delta^{\mathsf{cc}}_2)$, which works over the alphabet Γ_V described in Sect. 5.3.

States. Both the full set of states and the initial set of states for $\mathcal{A}^{\circlearrowleft}_{\mathsf{cc}}$ coincide with those of the word automaton $\mathcal{A}^{\dashrightarrow}_{\mathsf{exec}}$. That is, $Q^{\mathsf{cc}} = Q^{\mathsf{exec}}$ and $I^{\mathsf{cc}} = \{q_0^{\mathsf{exec}}\}$, where q_0^{exec} is the unique starting state of $\mathcal{A}^{\dashrightarrow}_{\mathsf{exec}}$.

Transitions. For intuition, consider the case when the automaton's control is in state q reading an internal tree node n with one child and which is labeled by $a =$ "**while**$(x = y)$". In the next step, the automaton simultaneously performs two transitions corresponding to two possibilities: entering the loop after assuming the guard "$x = y$" to be true and exiting the loop with the guard being false. In the first of these simultaneous transitions, the automaton moves to the left child $n{\cdot}L$, and its state changes to q'_1, where $q'_1 = \delta^{\mathsf{exec}}(q, \text{“}\mathbf{assume}(x = y)\text{”})$. In the second simultaneous transition, the automaton moves to the parent node $n{\cdot}U$ (searching for the next statement to execute, which follows the end of the loop) and changes its state to q'_2, where $q'_2 = \delta^{\mathsf{exec}}(q, \text{“}\mathbf{assume}(x \neq y)\text{”})$. We encode these two possibilities as a *conjunctive* transition of the two-way alternating automaton. That is, $\delta^{\mathsf{cc}}_1(q, m, a) = \big((q'_1, L) \wedge (q'_2, U)\big)$.

For every i, m, a, we have $\delta_i(q_{\mathsf{reject}}, m, a) = \bot$, where q_{reject} is the unique, absorbing rejecting state of $\mathcal{A}^{\dashrightarrow}_{\mathsf{exec}}$. Below we describe the transitions from all other states $q \neq q_{\mathsf{reject}}$. All transitions $\delta_i(q, m, a)$ not described below are $\bot$.

Transitions from the Root. At the root node, labeled by "**root**", the automaton transitions as follows:

$$\delta^{\mathsf{cc}}_1(q, m, \mathbf{root}) = \begin{cases} (q, L) \text{ if } m = D \\ \mathtt{true} \quad \text{otherwise} \end{cases}$$

A two-way tree automaton starts in the configuration where m is set to D. This means that in the very first step the automaton moves to the child node (direction L). If the automaton visits the root node in a subsequent step (marking the completion of an execution), then all transitions are enabled.

Transitions from Leaf Nodes. For a leaf node with label $a \in \Gamma_0$ and state q, the transition of the automaton is $\delta_0^{\mathsf{cc}}(q, D, a) = (\delta^{\mathsf{exec}}(q, a), U)$. That is, when the automaton visits a leaf node from the parent, it simulates reading a in $\mathcal{A}_{\mathsf{exec}}^{\cdots}$ and moves to the resulting state in the parent node.

Transitions from "**while**" ***Nodes.*** As described earlier, when reading a node labeled by "**while**$(x \sim y)$", where $\sim \in \{=, \neq\}$, the automaton simulates both the possibility of entering the loop body as well as the possibility of exiting the loop. This corresponds to a conjunctive transition:

$$\begin{aligned}\delta_1^{\mathsf{cc}}(q, m, \text{“}\mathbf{while}(x \sim y)\text{”}) &= (q', L) \wedge \big(q'', U\big) \\ \textit{where } q' &= \delta^{\mathsf{exec}}(q, \text{“}\mathbf{assume}(x \sim y)\text{”}) \\ \textit{and } q'' &= \delta^{\mathsf{exec}}(q, \text{“}\mathbf{assume}(x \not\sim y)\text{”})\end{aligned}$$

Above, $\not\sim$ refers to " = " when $\sim$ is " $\neq$ ", and vice versa. The first conjunct corresponds to the execution where the program enters the loop body (assuming the guard is true), and thus control moves to the left child of the current node, which corresponds to the loop body. The second conjunct corresponds to the execution where the loop guard is false and the automaton moves to the parent of the current tree node. Notice that, in both the conjuncts above, the direction in which the tree automaton moves does not depend on the last move m of the state. That is, no matter how the program arrives at a **while** statement, the automaton simulates both the possibilities of entering or exiting the loop body.

Transitions from "**ite**" ***Nodes.*** At a node labeled "**ite**$(x \sim y)$", when coming down the tree from the parent, the automaton simulates both branches of the conditional:

$$\begin{aligned}\delta_2^{\mathsf{cc}}(q, D, \text{“}\mathbf{ite}(x \sim y)\text{”}) &= (q', L) \wedge (q'', R) \\ \textit{where } q' &= \delta^{\mathsf{exec}}(q, \text{“}\mathbf{assume}(x \sim y)\text{”}) \\ \textit{and } q'' &= \delta^{\mathsf{exec}}(q, \text{“}\mathbf{assume}(x \not\sim y)\text{”})\end{aligned}$$

The first conjunct in the transition corresponds to simulating the word automaton on the condition $x \sim y$ and moving to the left child, i.e. the body of the **then** branch. Similarly, the second conjunct corresponds to simulating the word automaton on the negation of the condition and moving to the right child, i.e. the body of the **else** branch.

Now consider the case when the automaton moves *up* to an **ite** node from a child node. In this case, the automaton moves up to the parent node (having completed simulation of the **then** or **else** branch) and the state q remains unchanged:

$$\delta_2^{\mathsf{cc}}(q, m, \text{“}\mathbf{ite}(x \sim y)\text{”}) = (q, U) \qquad m \in \{U_L, U_R\}$$

Transitions from “**seq**” ***Nodes.*** In this case, the automaton moves either to the left child, the right child, or to the parent, depending on the last move. It does not change the state component. Formally,

$$\delta_2^{\mathsf{cc}}(q, m, \text{“}\mathbf{seq}\text{”}) = \begin{cases} (q, L) \text{ if } m = D \\ (q, R) \text{ if } m = U_L \\ (q, U) \text{ if } m = U_R \end{cases}$$

The above transitions match the straightforward semantics of sequencing two statements $s_1; s_2$. If the automaton visits from the parent node, it next moves to the left child to simulate s_1. When it finishes simulating s_1, it comes up from the left child and enters the right child to begin simulating s_2. Finally, when simulation of s_2 is complete, the automaton moves to the parent node, exiting the subtree.

The following lemma asserts the correctness of the automaton construction and states its complexity.

Lemma 2. *$\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc}}$ accepts the set of all program trees corresponding to correct, coherent programs. It has size $|\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc}}| = O(2^{\mathrm{poly}(|V|)})$, and can be constructed in $O(2^{\mathrm{poly}(|V|)})$ time.* □

5.5 Synthesis Procedure

The rest of the synthesis procedure goes as follows. We first construct a nondeterministic top-down tree automaton $\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc\text{-}td}}$ such that $L(\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc\text{-}td}}) = L(\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc}})$. An adaptation of results from [33,53] ensures that $\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc\text{-}td}}$ has size $|\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc\text{-}td}}| = O(2^{2^{\mathrm{poly}(|V|)}})$ and can be constructed in time $O(2^{2^{\mathrm{poly}(|V|)}})$. Next we construct a top-down nondeterministic tree automaton $\mathcal{A}^{\hat{\wedge}}$ such that $L(\mathcal{A}^{\hat{\wedge}}) = L(\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc\text{-}td}}) \cap L(\mathcal{A}^{\hat{\wedge}}_{\mathcal{G}}) = L(\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc}}) \cap L(\mathcal{A}^{\hat{\wedge}}_{\mathcal{G}})$, with size $|\mathcal{A}^{\hat{\wedge}}| = O(2^{2^{\mathrm{poly}(|V|)}} \cdot |\mathcal{G}|)$ and in time $O(|\mathcal{A}^{\hat{\wedge}}_{\mathsf{cc\text{-}td}}| \cdot |\mathcal{A}^{\hat{\wedge}}_{\mathcal{G}}|) = O(2^{2^{\mathrm{poly}(|V|)}} \cdot |\mathcal{G}|)$. Finally, checking emptiness of $\mathcal{A}^{\hat{\wedge}}$ can be done in time $O(|\mathcal{A}^{\hat{\wedge}}|) = O(2^{2^{\mathrm{poly}(|V|)}} \cdot |\mathcal{G}|)$. If non-empty, a program tree can be constructed.

This gives us the central upper bound result of the paper.

Theorem 6. *The grammar-restricted synthesis problem for uninterpreted coherent programs is decidable in* 2EXPTIME*, and in particular, in time doubly exponential in the number of variables and linear in the size of the input grammar. Furthermore, a tree automaton representing the set of* all *correct coherent programs that conform to the grammar can be constructed in the same time.* □

5.6 Matching Lower Bound

Our synthesis procedure is optimal. We prove a 2EXPTIME lower bound for the synthesis problem by reduction from the 2EXPTIME-hard acceptance problem of *alternating* Turing machines (ATMs) with exponential space bound [12]. Full details of the reduction can be found in [30].

Theorem 7. *The grammar-restricted synthesis problem for coherent uninterpreted programs is* 2EXPTIME*-hard.*

6 Further Results

In this section, we give results for variants of uninterpreted program synthesis in terms of transition systems, Boolean programs, and recursive programs.

6.1 Synthesizing Transition Systems

Here, rather than synthesizing programs from grammars, we consider instead the synthesis of transition systems whose executions must belong to a regular set. Our main result is that the synthesis problem in this case is EXPTIME-complete, in contrast to grammar-restricted program synthesis which is 2EXPTIME-complete.

Transition System Definition and Semantics. Let us fix a set of program variables V as before. We consider the following finite alphabet

$$\Sigma_V = \{\text{“}x := y\text{”}, \text{“}x := f(\overline{z})\text{”}, \text{“}\mathbf{assert}(\bot)\text{”}, \text{“}\mathbf{check}(x = y)\text{”} \mid x, y, \in V, \overline{z} \in V^r\}$$

Let us define $\Gamma_V \subseteq \Sigma_V$ to be the set of all elements of the form “$\mathbf{check}(x = y)$”, where $x, y \in V$. We refer to the elements of Γ_V as *check* letters.

A (deterministic) transition system TS over V is a tuple $(Q, q_0, H, \lambda, \delta)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $H \subseteq Q$ is the set of halting states, $\lambda : Q \to \Sigma_V$ is a labeling function such that for any $q \in Q$, if $\lambda(q) =$ “$\mathbf{assert}(\bot)$” then $q \in H$, and $\delta : (Q \setminus H) \to Q \cup (Q \times Q)$ is a transition function such that for any $q \in Q \setminus H$, $\delta(q) \in Q \times Q$ iff $\lambda(q) \in \Gamma_V$.

We define the semantics of a transition system using the set of executions that it generates. A *(partial) execution* π of a transition system $TS = (Q, q_0, H, \lambda, \delta)$ over variables V is a finite word over the induced execution alphabet Π_V (from Sect. 3) with the following property. If $\pi = a_0 a_1 \dots a_n$ with $n \geq 0$, then there exists a sequence of states $q_{j_0}, q_{j_1}, \dots, q_{j_n}$ with $q_{j_0} = q_0$ such that $(0 \leq i \leq n)$:

- If $\lambda(q_{j_i}) \notin \Gamma_V$ then $a_i = \lambda(q_{j_i})$, and if $i < n$ then $q_{j_{i+1}} = \delta(q_{j_i})$.
- Otherwise $\begin{cases} \text{either} & a_i = \text{“}\mathbf{assume}(x = y)\text{”} \text{ and } i < n \Rightarrow q_{j_{i+1}} = \delta(q_{j_i}) \downharpoonright_1, \\ \text{or} & a_i = \text{“}\mathbf{assume}(x \neq y)\text{”} \text{ and } i < n \Rightarrow q_{j_{i+1}} = \delta(q_{j_i}) \downharpoonright_2 \end{cases}$

In the above, we denote pair projection with $\lfloor$, i.e., $(t_1, t_2) \lfloor_i = t_i$, where $i \in \{1, 2\}$. A *complete execution* is an execution whose corresponding final state (q_n above) is in H. For any transition system TS, we denote the set of its executions by $\mathsf{Exec}(TS)$ and the set of its complete executions by $\mathsf{CompExec}(TS)$. The notions of *correctness* and *coherence* for transition systems are identical to their counterparts for programs.

The Transition System Synthesis Problem. We consider transition system specifications that place restrictions on executions (both partial and complete) using two regular languages S and R. Executions must belong to the first language S (which is prefix-closed) and all complete executions must belong to the second language R. A specification is given as two deterministic automata $\mathcal{A}_S^{\cdots}$ and $\mathcal{A}_R^{\cdots}$ over executions, where $L(\mathcal{A}_S^{\cdots}) = S$ and $L(\mathcal{A}_R^{\cdots}) = R$. For a transition system TS and specification automata $\mathcal{A}_S^{\cdots}$ and $\mathcal{A}_R^{\cdots}$, whenever $\mathsf{Exec}(TS) \subseteq L(\mathcal{A}_S^{\cdots})$ and $\mathsf{CompExec}(TS) \subseteq L(\mathcal{A}_R^{\cdots})$ we say that TS satisfies its (syntactic) specification. Note that this need not entail correctness of TS. Splitting the specification into partial executions S and complete executions R allows us, among other things, to constrain the executions of non-halting transition systems.

Definition 3 (Transition System Realizability and Synthesis). *Given a finite set of program variables V and deterministic specification automata $\mathcal{A}_S^{\cdots}$ (prefix-closed) and $\mathcal{A}_R^{\cdots}$ over the execution alphabet Π_V, decide if there is a* correct*, coherent transition system TS over V that satisfies the specification. Furthermore, produce one if it exists.*

Since programs are readily translated to transition systems (of similar size), the transition system synthesis problem seems, at first glance, to be a problem that ought to have similar complexity. However, as we show, it is crucially different in that it allows the synthesized transition system to have *complete information* of past commands executed at any point. We will observe in this section that the transition system synthesis problem is EXPTIME-complete.

To see the difference between program and transition system synthesis, consider program skeleton P from Example 2 in Sect. 2. The problem is to fill the hole in P with either $\mathtt{y} := \mathtt{T}$ or $\mathtt{y} := \mathtt{F}$. Observe that when P executes, there are *two* different executions that lead to the hole. In grammar-restricted program synthesis, the hole must be filled by a sub-program that is executed *no matter how the hole is reached*, and hence no such program exists. However, when we model this problem in the setting of transition systems, the synthesizer is able to produce transitions that depend on how the hole is reached. In other words, it does not fill the hole in P with *uniform* code. In this sense, in grammar-restricted program synthesis, programs have *incomplete information* of the past. We crucially exploited this difference in the proof of 2EXPTIME-hardness for grammar-restricted program synthesis (see [30]). No such incomplete information can be enforced by regular execution specifications in transition system synthesis, and indeed the problem turns out to be easier: transition system realizability and synthesis are EXPTIME-complete.

Theorem 8. *Transition system realizability is decidable in time exponential in the number of program variables and polynomial in the size of the automata* $\mathcal{A}_S^{\cdots}$ *and* $\mathcal{A}_R^{\cdots}$*. Furthermore, the problem is* EXPTIME*-complete. When realizable, within the same time bounds we can construct a correct, coherent transition system whose partial and complete executions are in* $L(\mathcal{A}_S^{\cdots})$ *and* $L(\mathcal{A}_R^{\cdots})$*, respectively.*

6.2 Synthesizing Boolean Programs

Here we observe corollaries of our results when applied to the more restricted problem of synthesizing Boolean programs.

In Boolean program synthesis we interpret variables in programs over the Boolean domain $\{T, F\}$, and we disallow computations of uninterpreted functions and the checking of uninterpreted relations. Standard Boolean functions like $\wedge$ and $\neg$ are instead allowed, but note that these can be modeled using conditional statements. We allow for *nondeterminism* with a special assignment "$b := *$", which assigns b nondeterministically to T or F. As usual, a program is correct when it satisfies all its assertions.

Synthesis of Boolean programs can be reduced to uninterpreted program synthesis using two special constants T and F. Each nondeterministic assignment is modeled by computing a `next` function on successive nodes of a linked list, accessing a nondeterministic value by computing `key` on the current node, and assuming the result is either T or F. Since uninterpreted programs must satisfy assertions in all models, this indeed captures nondeterministic assignment. Further, every term ever computed in such a program is equivalent to T or F (by virtue of the interleaved **assume** statements), making the resulting program coherent. The 2EXPTIME upper bound for Boolean program synthesis now follows from Theorem 6. We further show that, perhaps surprisingly, the 2EXPTIME lower bound from Sect. 5 can be adapted to prove 2EXPTIME-hardness of Boolean program synthesis.

Theorem 9. *The grammar-restricted synthesis problem for Boolean programs is* 2EXPTIME*-complete, and can be solved in time doubly-exponential in the number of variables and linear in the size of the input grammar.* □

Thus synthesis for coherent uninterpreted programs is no more complex than Boolean program synthesis, establishing decidability and complexity of a problem which has found wide use in practice—for instance, the synthesis tool SKETCH solves precisely this problem, as it models integers using a small number of bits and allows grammars to restrict programs with holes.

6.3 Synthesizing Recursive Programs

We extend the positive result of Sect. 5 to synthesize coherent recursive programs. The setup for the problem is very similar. Given a grammar that identifies a class of recursive programs, the goal is to determine if there is a program in the grammar that is coherent and correct.

The syntax of recursive programs is similar to the non-recursive case, and we refer the reader to [30] for details. In essence, programs are extended with a new function call construct. Proofs are similar in structure to the non-recursive case, with the added challenge of needing to account for recursive function calls and the fact that $\mathcal{A}^{\cdots}_{\mathsf{exec}}$ becomes a (visibly) pushdown automaton rather than a standard finite automaton. This gives a 2EXPTIME algorithm for synthesizing recursive programs; a matching lower bound follows from the non-recursive case.

Theorem 10. *The grammar-restricted synthesis problem for uninterpreted coherent* recursive *programs is* 2EXPTIME*-complete. The algorithm is doubly exponential in the number of program variables and linear in the size of the input grammar. Furthermore, a tree automaton representing the set of all correct, coherent recursive programs that conform to the grammar can be constructed in the same time.*

7 Related Work

The automata and game-theoretic approaches to synthesis date back to a problem proposed by Church [13], after which a rich theory emerged [9,18,32,48]. The problems considered in this line of work typically deal with a system reacting to an environment interactively using a finite set of signals over an infinite number of rounds. Tree automata over infinite *trees*, representing strategies, with various infinitary acceptance conditions (Büchi, Rabin, Muller, parity) emerged as a uniform technique to solve such synthesis problems against temporal logic specifications with optimal complexity bounds [31,38,44,45]. In this paper, we use an alternative approach from [35] that works on *finite* program trees, using two-way traversals to simulate iteration. The work in [35], however, uses such representations to solve synthesis problems for programs over a fixed finite set of Boolean variables and against LTL specifications. In this work we use it to synthesize coherent programs that have finitely many variables working over infinite domains endowed with functions and relations.

While decidability results for program synthesis beyond finite data domains are uncommon, we do know of some results of this kind. First, there are decidability results known for synthesis of tranducers with registers [29]. Transducers interactively read a stream of inputs and emit a stream of outputs. Finite-state tranducers can be endowed with a set of registers for storing inputs and doing only equality/disequality comparisons on future inputs. Synthesis of such transducers for temporal logic specifications is known to be decidable. Note that, although the data domain is infinite, there are no functions or relations on data (other than equality), making it a much more restricted class (and grammar-based approaches for syntactically restricting transducers has not been studied). Indeed, with uninterpreted functions and relations, the synthesis problem is undecidable (Theorem 1), with decidability only for coherent programs. In [11], the authors study the problem of synthesizing uninterpreted terms from a grammar that satisfy a first-order specification. They give various decidability and

undecidability results. In contrast, our results are for programs with conditionals and iteration (but restricted to coherent programs) and for specifications using assertions in code.

Another setting with a decidable synthesis result over unbounded domains is work on strategy synthesis for linear arithmetic *satisfiability* games [17]. There it is shown that for a satisfiability game, in which two players (SAT and UNSAT) play to prove a formula is satisfiable (where the formula is interpreted over the theory of linear rational arithmetic), if the SAT player has a winning strategy then a strategy can be synthesized. Though the data domain (rationals) is infinite, the game consists of a finite set of interactions and hence has no need for recursion. The authors also consider reachability games where the number of rounds can be unbounded, but present only sound and incomplete results, as checking who wins in such reachability games is undecidable.

Tree automata techniques for accepting finite parse trees of programs was explored in [37] for synthesizing reactive programs with variables over finite domains. In more recent work, automata on finite trees have been explored for synthesizing data completion scripts from input-output examples [55], for accepting programs that are verifiable using abstract interpretations [54], and for relational program synthesis [56].

The work in [36] explores a decidable logic with $\exists^*\forall^*$ prefixes that can be used to encode synthesis problems with background theories like arithmetic. However, encoding program synthesis in this logic only expresses programs of finite size. Another recent paper [27] explores sound (but incomplete) techniques for showing unrealizability of syntax-guided synthesis problems.

8 Conclusions

We presented foundational results on synthesizing coherent programs with uninterpreted functions and relations. To the best of our knowledge, this is the first natural decidable program synthesis problem for programs of arbitrary size which have iteration/recursion, and which work over infinite domains.

The field of program synthesis lacks theoretical results, and especially decidability results. We believe our results to be the first of their kind to fill this lacuna, and we find this paper exciting because it bridges the worlds of program synthesis and the rich classical synthesis frameworks of systems over finite domains using tree automata [9,18,32,48]. We believe this link could revitalize both domains with new techniques and applications.

Turning to practical applications of our results, several questions require exploration in future work. First, one might question the utility of programs that verify only with respect to uninterpreted data domains. Recent work [10] has shown that verifying programs using uninterpreted abstractions can be extremely effective in practice for proving programs correct. Also, recent work by Mathur et al. [40] explores ways to add *axioms* (such as commutativity of functions, axioms regarding partial orders, etc.) and yet preserve decidability of verification. The methods used therein are compatible with our technique, and we

believe our results can be extended smoothly to their decidable settings. A more elaborate way to bring in complex theories (like arithmetic) would be to marry our technique with the *iterative* automata-based software verification technique pioneered by work behind the ULTIMATE tool [23–26]; this won't yield decidable synthesis, but still could result in *complete* synthesis procedures.

The second concern for practicality is the coherence restriction. There is recent work by Mathur et al. [41] that shows single-pass heap-manipulating programs respect a (suitably adapted) notion of coherence. Adapting our technique to this setting seems feasible, and this would give an interesting application of our work. Finally, it is important to build an implementation of our procedure in a tool that exploits pragmatic techniques for constructing tree automata, and the techniques pursued in [54–56] hold promise.

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 40, pp. 1–25. IOS Press (2015)
2. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing, STOC 2004, pp. 202–211. ACM, New York (2004). https://doi.org/10.1145/1007352.1007390
3. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM **56**(3), 16:1–16:43 (2009). https://doi.org/10.1145/1516512.1516518
4. Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Commun. ACM **61**(12), 84–93 (2018). https://doi.org/10.1145/3208071
5. Bauer-Mengelberg, S.: über die vollständigkeit des logikkalküls. J. Symb. Log. **55**(1), 341–342 (1990). https://doi.org/10.2307/2274974
6. Bloem, R., Galler, S.J., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: hardware from PSL. Electr. Notes Theor. Comput. Sci. **190**(4), 3–16 (2007). https://doi.org/10.1016/j.entcs.2007.09.004
7. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012). https://doi.org/10.1016/j.jcss.2011.08.007
8. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74113-8
9. Buchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Trans. Am. Math. Soc. **138**, 295–311 (1969). https://doi.org/10.2307/1994916
10. Bueno, D., Sakallah, K.A.: euforia: complete software model checking with uninterpreted functions. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 363–385. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_17
11. Caulfield, B., Rabe, M.N., Seshia, S.A., Tripakis, S.: What's decidable about syntax-guided synthesis? CoRR abs/1510.08393 (2015)
12. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM **28**(1), 114–133 (1981). https://doi.org/10.1145/322234.322243
13. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis. Summaries of talks presented at the Summer Institute for Symbolic Logic Cornell University, 1957, 2nd edn., J. Symb. Log. **28**(4), 30–50. 3a–45a. (1960)

14. Comon, H., et al.: Tree automata techniques and applications (2007). https://tata.gforge.inria.fr. Accessed 29 Jun 2020
15. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Trans. Inf. Theory **29**(2), 198–208 (1983). https://doi.org/10.1109/TIT.1983.1056650
16. Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Multiset rewriting and the complexity of bounded security protocols. J. Comput. Secur. **12**(2), 247–311 (2004). https://doi.org/10.3233/JCS-2004-12203
17. Farzan, A., Kincaid, Z.: Strategy synthesis for linear arithmetic games. PACMPL **2**(POPL), 61:1–61:30 (2018). https://doi.org/10.1145/3158149
18. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]. Lecture Notes in Computer Science, vol. 2500. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36387-4
19. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL, pp. 317–330. ACM (2011). https://doi.org/10.1145/1925844.1926423
20. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Commun. ACM **55**(8), 97–105 (2012). https://doi.org/10.1145/2240236.2240260
21. Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S.H., Schmid, U., Zorn, B.G.: Inductive programming meets the real world. Commun. ACM **58**(11), 90–99 (2015). https://doi.org/10.1145/2736282
22. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1–2), 1–119 (2017)
23. Heizmann, M., et al.: Ultimate automizer with smtinterpol. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 641–643. Springer, Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_53
24. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_7
25. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 471–482. ACM, New York (2010). https://doi.org/10.1145/1706299.1706353
26. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_2
27. Hu, Q., Breck, J., Cyphert, J., D'Antoni, L., Reps, T.: Proving unrealizability for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 335–352. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_18
28. Jha, S., Seshia, S.A.: A theory of formal synthesis via inductive learning. Acta Inf. **54**(7), 693–726 (2017). https://doi.org/10.1007/s00236-017-0294-5
29. Khalimov, A., Maderbacher, B., Bloem, R.: Bounded synthesis of register transducers. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 494–510. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_29
30. Krogmeier, P., Mathur, U., Murali, A., Madhusudan, P., Viswanathan, M.: Decidable synthesis of programs with uninterpreted functions. CoRR abs/1910.09744 (2019). http://arxiv.org/abs/1910.09744

31. Kupferman, O., Madhusudan, P., Thiagarajan, P.S., Vardi, M.Y.: Open systems in reactive environments: control and synthesis. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 92–107. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_9
32. Kupferman, O., Piterman, N., Vardi, M.Y.: An automata-theoretic approach to infinite-state systems. In: Manna, Z., Peled, D.A. (eds.) Time for Verification. LNCS, vol. 6200, pp. 202–259. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13754-9_11
33. Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to reasoning about infinite-state systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 36–52. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_7
34. Löding, C., Madhusudan, P., Neider, D.: Abstract learning frameworks for synthesis. In: Chechik, M., Raskin, J.F. (eds.) LTACAS 2016. LNCS, vol. 9636, pp. 167–185. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_10
35. Madhusudan, P.: Synthesizing reactive programs. In: CSL. LIPIcs, vol. 12, pp. 428–442. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011). https://doi.org/10.4230/LIPIcs.CSL.2011.428
36. Madhusudan, P., Mathur, U., Saha, S., Viswanathan, M.: A decidable fragment of second order logic with applications to synthesis. In: Ghica, D., Jung, A. (eds.) 27th EACSL Annual Conference on Computer Science Logic (CSL 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 119, pp. 31:1–31:19. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2018). https://doi.org/10.4230/LIPIcs.CSL.2018.31
37. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 283–294. ACM, New York (2011). https://doi.org/10.1145/1926385.1926419
38. Madhusudan, P., Thiagarajan, P.S.: Distributed controller synthesis for local specifications. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 396–407. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48224-5_33
39. Mathur, U., Madhusudan, P., Viswanathan, M.: Decidable verification of uninterpreted programs. Proc. ACM Program. Lang. **3**(POPL), 46:1–46:29 (2019). https://doi.org/10.1145/3290359
40. Mathur, U., Madhusudan, P., Viswanathan, M.: What's decidable about program verification modulo axioms? In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12079, pp. 158–177. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_10
41. Mathur, U., Murali, A., Krogmeier, P., Madhusudan, P., Viswanathan, M.: Deciding memory safety for single-pass heap-manipulating programs. Proc. ACM Program. Lang. **4**(POPL), 1–29 (2019). https://doi.org/10.1145/3371103
42. Müller-Olm, M., Rüthing, O., Seidl, H.: Checking herbrand equalities and beyond. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 79–96. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_6
43. Muscholl, A., Walukiewicz, I.: Distributed synthesis for acyclic architectures. In: FSTTCS. LIPIcs, vol. 29, pp. 639–651. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014). https://doi.org/10.4230/LIPIcs.FSTTCS.2014.639
44. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190. ACM Press (1989). https://doi.org/10.1145/75277.75293

45. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: FOCS, pp. 746–757. IEEE Computer Society (1990). https://doi.org/10.1109/FSCS.1990.89597
46. Post, E.L.: A variant of a recursively unsolvable problem. Bull. Amer. Math. Soc. **52**(4), 264–268 (1946). https://doi.org/10.1090/S0002-9904-1946-08555-9
47. Qiu, X., Solar-Lezama, A.: Natural synthesis of provably-correct data-structure manipulations. PACMPL **1**(OOPSLA), 65:1–65:28 (2017). https://doi.org/10.1145/3133889
48. Rabin, M.O.: Automata on Infinite Objects and Church's Problem. American Mathematical Society, Boston (1972)
49. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. SIGPLAN Not. **48**(6), 15–26 (2013). https://doi.org/10.1145/2499370.2462195
50. Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. **15**(5), 475–495 (2013). https://doi.org/10.1007/s10009-012-0249-7
51. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415. ACM (2006). https://doi.org/10.1145/1168857.1168907
52. SyGuS: Syntax guided synthesis. https://sygus.org/
53. Vardi, M.Y.: Reasoning about the past with two-way automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 628–641. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055090
54. Wang, X., Dillig, I., Singh, R.: Program synthesis using abstraction refinement. Proc. ACM Program. Lang. **2**(POPL), 63:1–63:30 (2017). https://doi.org/10.1145/3158151
55. Wang, X., Gulwani, S., Singh, R.: FIDEX: filtering spreadsheet data using examples. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, pp. 195–213. ACM, New York (2016). https://doi.org/10.1145/2983990.2984030
56. Wang, Y., Wang, X., Dillig, I.: Relational program synthesis. Proc. ACM Program. Lang. **2**(OOPSLA), 155:1–155:27 (2018). https://doi.org/10.1145/3276525

Must Fault Localization for Program Repair

Bat-Chen Rothenberg and Orna Grumberg(✉)

Technion - Israel Institute of Technology, Haifa, Israel
{batg,orna}@cs.technion.ac.il

Abstract. This work is concerned with fault localization for automated program repair.

We define a novel concept of a *must* location set. Intuitively, such a set includes at least one program location from every repair for a bug. Thus, it is impossible to fix the bug without changing at least one location from this set. A fault localization technique is considered a *must* algorithm if it returns a must location set for every buggy program and every bug in the program. We show that some traditional fault localization techniques are not must.

We observe that the notion of must fault localization depends on the chosen repair scheme, which identifies the changes that can be applied to program statements as part of a repair. We develop a new algorithm for fault localization and prove that it is *must* with respect to commonly used schemes in automated program repair.

We incorporate the new fault localization technique into an existing mutation-based program repair algorithm. We exploit it in order to prune the search space when a buggy mutated program has been generated. Our experiments show that must fault localization is able to significantly speed-up the repair process, without losing any of the potential repairs.

1 Introduction

Fault localization and automated program repair have long been combined. Traditionally, given a buggy program, fault localization suggests locations in the program that might be the cause of the bug. Repair then attempts to change those suspicious locations in order to eliminate the bug.

Bad fault localization may cause a miss of potential repairs, if it is too restrictive, or cause an extra work, if it is too permissive. Studies have shown that for test-based repair imprecise fault localizations happen very often in practice [27]. This identifies the need for fault localization that can narrow down the space of candidates while still promising not to lose potential causes for a bug.

In this work, we define the concept of a *must* location set. Intuitively, such a set includes at least one location from every repair for the bug. Thus, it *must* be

This research was partially supported by the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau and partially by the Israel Science Foundation.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12225, pp. 658–680, 2020.
https://doi.org/10.1007/978-3-030-53291-8_33

used for repair. In other words, **it is impossible to fix the bug using only locations outside this set**. A fault localization technique is considered a *must* algorithm if it returns a must location set for every buggy program and every bug in the program.

To demonstrate the importance of the *must* notion, consider the program in Fig. 1 for computing the absolute value of a variable `x`. The program is buggy since the assertion in location 4 is violated when initially `x = -1`. Intuitively, a good repair would replace the condition `(x < -1)` in location 2 with condition `x <= -1`. Our must fault localization, defined formally in the paper, will include location 2 in the must location set. In contrast, the fault localization techniques defined for instance in [14,21] do not include 2 in their location sets: They are not must and may miss optional repairs.

Our first observation regarding must notions is that their definition should take into account the *repair scheme* under consideration. A repair scheme identifies the changes that can be applied to program statements as part of a repair. A scheme can allow, for instance, certain syntactic changes in a condition (e.g. replacing `<` with `>`) or in the right-hand-side expression of an assignment (e.g. replacing `+` by `-`). A particular location set can be a *must* set using one scheme, but non-*must* using another. We further discuss this observation when presenting our formal definition of a must fault localization.

The setting of our work is as follows. Our approach is formula-based rather than test-based. We handle simple C-programs, with specification given as assertions in the code. Similarly to bounded model checking tools (e.g. [8]), the program and the negated specification are translated to a set of constraints, whose conjunction forms the *program formula*. This formula is satisfiable if and only if the program violates an assertion, in which case a satisfying assignment (also called a *model*) is returned.

We focus on a simple repair scheme of syntactic changes, as described above. We assume that the user prefers repairs that are as close to the original program as possible and will want to get several repair suggestions. Thus, we return *all minimal repairs* (minimal in the number of changes applied to the program code).

Once the notion of must fault localization is defined, we develop a new algorithm for fault localization and prove that it is *must* with respect to syntactic mutation schemes. The input to the algorithm is a program formula φ and a model μ for φ, representing a buggy execution of the program. Our approach is based on a dynamic-slicing-like algorithm that computes dependencies.

For a variable v in φ, its slice F is computed based on dynamic dependencies among variables in φ, whose values influence the value of v in μ. Informally, F is a must location set that contains all assignment to the variables that v depends upon. Some assignment from F thus must be changed in order to eliminate the bug associated with μ.

We incorporated the new fault localization technique into an existing mutation-based program repair algorithm [38]. In [38], the repair scheme is based on a predefined set of mutations. Given a buggy program P, the goal of the algorithm is to return all minimal repairs for P. The algorithm goes through

iterations of generate-validate, where the generate part produces a mutated program of P and the validate part checks whether it is bounded-correct. The bottleneck of the algorithm is the size of the search space, consisting of all possible mutated programs of P. In [38], the search space has been pruned when the generated mutated program has been successfully validated. No pruning has been applied otherwise.

In this work, we exploit our novel *must* fault localization in order to prune the search space when a buggy mutated program P' has been generated (i.e. validation failed). In this case, we compute the *must* location set F of P'. We can now prune from the search space any mutated program whose F locations are identical to those of P'. This is because, by the property of *must* location set, it is guaranteed that the bug cannot be repaired without changing a location in F. Thus, a large set of buggy mutated programs is pruned, without the need for additional validation and without losing any minimally repaired program. It should be noted that the smaller F is, the larger the pruned set is. Our experimental results confirm the effectiveness of this pruning by showing significant speedups.

To summarize, the contributions of this work are:

1. We define a novel notion of *must* fault localization with respect to a repair scheme. We show that many of the formula-based techniques are not must.
2. We present a novel fault localization technique and prove that it is *must* for the scheme of syntactic mutations. Our technique also has other advantages, such as low-complexity and incrementality.
3. We show how our new fault localization technique can be incorporated into an existing mutation-based program repair algorithm for pruning its search space. The technique is applied iteratively, whenever a generated mutated program is found to be incorrect.
4. We implemented the algorithm of repair with fault localization as part of the open source tool AllRepair. Our experimental results show that fault-localization is able to significantly speed-up the repair process, without losing any of the potential repairs.

2 Motivating Example

Figure 1 presents a simple program for computing the absolute value of a variable `x`. The result is computed in the variable `abs`, and the specification states, using an assertion on line 4, that in the end `abs` should always be non-negative. Unfortunately, the program has a bug. The true branch of the if is intended to flip the sign of `x` whenever `x` is negative, but it accidentally misses the case where `x` is -1. As a result, if `x` is -1, the wrong branch of the if is taken, and the assertion is reached with `abs` $= -1$, which causes a violation.

```
   procedure absValue(x)
1: abs := x
2: if x < -1 then
3:     abs := -x
4: assert (abs >= 0)
```

Fig. 1. A buggy program

Clearly, it is desirable that line number 2 be returned when running fault localization on this bug, as a human written repair is likely to change the condition on this line from `x < -1` to `x <= -1` or `x < 0`. But, as we will show next, some of the existing formula-based fault localization techniques do not include this line in their result.

The error trace representing the bug for input $I = \{\texttt{x} \leftarrow -1\}$ is $\pi =< 1, 2, 4 >$ (this is the sequence of program locations visited when executing the program on I). The MAX-SAT-based fault localization technique of [21] and the error-invariant-based technique of [14] use a formula called the *extended trace formula* in order to find faulty statements along the error trace. The extended trace formula for the bug in question is

$$\underbrace{(x = -1)}_{\text{Input}} \wedge \underbrace{(abs = x) \wedge (x \geq -1)}_{\text{Computation}} \wedge \underbrace{(abs \geq 0)}_{\text{Assertion}}$$

This formula encodes three things: a) that the input remains I, b) that the computation is as the trace dictates, and, c) that the assertion holds at the end. Therefore, the formula is unsatisfiable. Both [21] and [14] intuitively look for explanations of its unsatisfiability, and therefore decide that the statement $(x \geq -1)$ on line 2 is irrelevant; The formula remains unsatisfiable even if the constraint $(x \geq -1)$ is removed.

Even the method of [6], which suggests a flow-sensitive encoding of the extended trace formula, with the goal of including all statements affecting control-flow decisions that are relevant to the bug, classifies the statement on line 2 as irrelevant. This is because the error trace does not include any location from the body of the branch that was taken (in our case it is the `else` branch, which is empty), in which case the flow-sensitive formula remains identical to the traditional formula.

The dynamic slicing method of [2,23] also fails to include line 2 in its result. This method computes the set of statements influencing the evaluation of the assertion along the trace, using data and control dependency relations. A statement st_1 is data dependent on st_2 iff st_1 uses a variable x, and st_2 is the last to assign a value to x along the trace. In our example, the assertion on line 4 is data dependent only on the statement in line 1, which in itself is not data dependent on any other statement. A statement st_1 is control dependent on a conditional statement st_2 iff st_1 is inside the body of either branch of st_2. None of the statements along our error trace is control dependent on another statement. The slice, which is the set of lines returned, is computed using the transitive closure of these relations. Thus, for our example, only line 1 is part of the slice.

In this example, we have seen how many different fault localization techniques fail to include a statement that is relevant, i.e., where a modification could be made for the bug to be fixed. In contrast, the set of locations returned by our technique for this example is $\{1, 2\}$. The fact that our technique includes line 2 is not a coincidence: We show that, intuitively, whenever a repair can be made by making changes to a single line, this line *must* be included in the result.

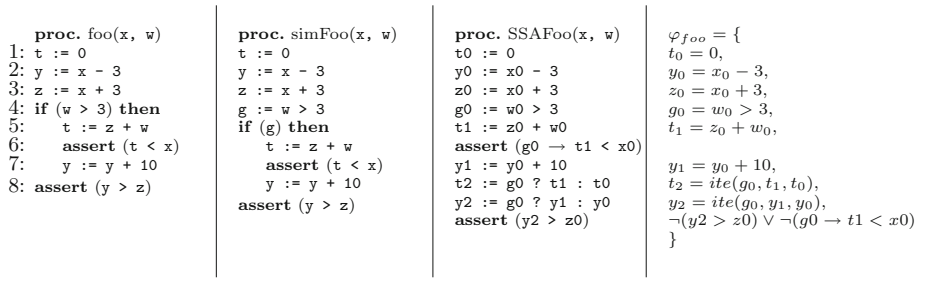


Fig. 2. Example of the translation process of a simple program

In general, whenever a repair can be made by making changes to a set of lines, at least one of them must be included in the result.

3 Preliminaries

3.1 Programs and Error Traces

For our purposes, a *program* is a sequential program composed of standard statements: assignments, conditionals, loops and function calls, all with their standard semantics. Each statement is located at a certain *location* (or *line*) l_i, and all statements are defined over the set of program variables X.

In addition to the standard statements, a program may also contain *assume* statements of the form `assume(bexpr)`, and *assert* statements of the form `assert(bexpr)`. In both cases `bexpr` is a boolean expression over X. If an assume or an assert statement is located in l_i, execution of the program stops whenever location l_i is reached in a state where `bexpr` is evaluated to false. In the case of an assertion, this early termination has the special name *assertion violation*, and it is an indication that an error has occurred.

A program P has a *bug on input* I if an assertion violation occurs during the execution of P on I. Otherwise, the program is *correct for* I.[1] Whenever P has a bug on I, this bug is associated with an *error trace*, which is the sequence of statements visited during the execution of P on I.

3.2 From Programs to Program Formulas

In this section we explain how a program is translated into a set of constraints, whose conjunction constitutes the program formula. In addition to constraints representing assignments and conditionals, such a formula includes constraints representing assumptions and a constraint representing the negated conjunction of all assertions. Thus, a satisfying assignment (a *model*) of the program formula

[1] Alternatively, one could assume to know the desired output of the program for I and define a bug on I as a case where the program outputs the wrong value for I.

represents an execution of the program that satisfies all assumption but violates at least one assertion. Such an execution is a *counterexample*.

The translation, following [8], goes through four stages. We refer to the example in Fig. 2 to demonstrate certain steps.

1. Simplification: Complex constructs of the language are replaced with equivalent simpler ones. Also, branch conditions are replaced with fresh boolean variables. In the example, the `if` condition `(w > 3)` is assigned to a fresh boolean variable `g`. Branching is then done based on the value of `g`, instead of `(w > 3)`.
2. Unwinding: The body of each loop and each function is inlined wb times. The set of executions of the new program is called the wb-executions of P.
3. Conversion to SSA: The program is converted to static single assignment (SSA) form, which means that each variable in the new program is assigned at most once. This is done by replacing all variables with indexed variables, and increasing the index of a variable whenever it appears on the left-hand-side of an assignment. In the example, the first assignment to `t` is replaced by an assignment to `t0` and the second, by an assignment to `t1`. Since `t` is assigned inside a conditional statement and is used after the statement, the if-then-else assignment `t2 := g0?t1:t0` is inserted in order to determine which copy of `t` should be used after the conditional statement. These special if-then-else assignments are called *Φ-assignments*. In the example, there is also a Φ-assignment for `y` (`y2=g0?y1:y0`).
   Note that, assertions are also expressed by means of indexed variables. The specific indices in the assertion indicate the location in the execution in which the assertion is checked. In addition, if an assumption or an assertion is located within an `if` statement with branch condition g, then it is implied by g if it is within the `then` part of the `if` and is implied by $\neg g$, if it is within the `else` part. In the example, `assert (t < x)` is encoded by $(g_0 \rightarrow t_1 < x_0)$.
4. Conversion to SMT constraints: Once the program is in SSA form, conversion to SMT is straightforward: An assignment `x:=e` is converted to the constraint $x = e$; A Φ-assignment `x:= b?x1:x2` is converted to the constraint $(x = ite(b, x_1, x_2))$, which is an abbreviation of $((b \wedge x = x_1) \vee (\neg b \wedge x = x_2))$; An assume statement `assume(bexpr)` is converted to the constraint `bexpr`, and an assert statement `assert(bexpr)` is converted to the constraint ¬`bexpr` (since a model of the SMT formula should correspond to an assertion violation).
   If the program includes several assertions, then they are converted to one constraint, representing the negation of their conjunction. In the example, the two assertions are converted to the following constraint:

$$\neg(y2 > z0) \vee \neg(g0 \rightarrow t1 < x0).$$

   We say that a constraint *encodes* the statement it came from and we partition constraints into three sets, S_{assign}, S_{phi} and S_{demand}, based on what they encode. S_{assign} contains constraints encoding assignments, including those originated from assigning a fresh boolean variable with a branching condition;

S_{phi} - encoding Φ-assignments; and S_{demand} - encoding demands from assert and assume statements. In particular, it encodes the negated conjunction of all assertions.

The triple $(S_{assign}, S_{phi}, S_{demand})$ is called a *program constraint set*. The program constraint set we get from a program P when using wb as an unwinding bound is denoted CS_P^{wb}. The *program formula* φ_P^{wb}, is the conjunction of all constraints in all three sets of CS_P^{wb}:

$$\varphi_P^{wb} = (\bigwedge_{s \in S_{assign}} s) \wedge (\bigwedge_{s \in S_{phi}} s) \wedge (\bigwedge_{s \in S_{demand}} s).$$

Theorem 1 ([9]). *A program P is wb-violation free iff the formula φ_P^{wb} is unsatisfiable.*

For simplicity of notation, in the rest of the paper we omit the superscript wb.

Since the program formula is the result of translating an SSA program, the formula is defined over indexed variables. Further, each constraint in S_{assign} corresponds to the single variable, which is assigned in the statement encoded by the constraint.

4 Must Fault Localization

In this section, we precisely define when a location should be considered relevant for a bug. This definition is motivated by a repair perspective, taking into account which changes can be made to statements in order to repair a bug.

In order to define the changes allowed, we use repair schemes. A *repair scheme* $\mathcal{S}$ is a function from statements to sets of statements. An $\mathcal{S}$*-patch* for a program P is a set of pairs of location and statement $\{(l_1, st_1^r), \cdots, (l_k, st_k^r)\}$, for which the following holds: for all $1 \leq i \leq k$, let st_i be the statement in location l_i in P, then $st_i^r \in \mathcal{S}(st_i)$. The patch is said to be *defined over* the set of locations $\{l_1, \cdots, l_k\}$. Applying an $\mathcal{S}$*-patch* τ to a program P means replacing for every location l_i in τ, the statement st_i with st_i^r. This results in an $\mathcal{S}$*-patched* program of P. The set of all $\mathcal{S}$*-patched* programs created from a program P is the $\mathcal{S}$*-search space* of P.

Let P be a program with a bug on input I, and $\mathcal{S}$ be a repair scheme. An $\mathcal{S}$*-repair* for I is an $\mathcal{S}$-patched program that is correct for I. An $\mathcal{S}$*-repairable set* is a set of locations F such that there exists an $\mathcal{S}$-repair defined over F. An $\mathcal{S}$-repairable set is *minimal* if removing any location from it makes it no longer an $\mathcal{S}$-repairable set. A location is $\mathcal{S}$*-relevant* if it is a part of a minimal $\mathcal{S}$-repairable set.[2]

In this paper, we focus on two repair schemes that are frequently used for automated program repair: the arbitrary scheme ($\mathcal{S}_{arb}$) and the mutation scheme ($\mathcal{S}_{mut}$). Both schemes only manipulate program expressions, but the

[2] We sometimes omit $\mathcal{S}$ from notations where $\mathcal{S}$ is clear from context.

mutation scheme is more restrictive than the arbitrary scheme: $\mathcal{S}_{arb}(st)$ is the set of all options to replace the expression of st[3] with an arbitrary expression, while $\mathcal{S}_{mut}(st)$ only contains statements where the expression in st is mutated according to a set of simple syntactic rules. The rules we consider are replacing a `+` operator with a `-` operator, and vice versa, replacing a `<` operator with a `>` operator, and vice versa, and increasing or decreasing a numerical constant by 1.[4]

Example 1. In this example we demonstrate how different repair schemes define different sets of relevant locations. Consider again the foo program from Fig. 2. This program has a bug on input $I = \mathtt{x} \leftarrow 0, \mathtt{w} \leftarrow 0$. The error trace associated with the bug is $\langle 1, 2, 3, 4, 8 \rangle$ (the assertion on line 8 is violated).

The location set $\{3, 4\}$ is a minimal $\mathcal{S}_{mut}$-repairable set: It is an $\mathcal{S}_{mut}$-repairable set because applying the $\mathcal{S}_{mut}$-patch $\{(3, \mathtt{z:=x-3}), (4, \mathtt{w<3})\}$, results in an $\mathcal{S}_{mut}$-patched program that is correct for I. This set is also minimal, because none of the $\mathcal{S}_{mut}$-patches defined over $\{3\}$ or $\{4\}$ alone is an $\mathcal{S}_{mut}$-repair for I: Each one of the $\mathcal{S}_{mut}$-patches $\{(3, \mathtt{z:=x-3})\}, \{(3, \mathtt{z:=x+4})\}, \{(3, \mathtt{z:=x+2})\}$, $\{(4, \mathtt{w<3})\}, \{(4, \mathtt{w>4})\}, \{(4, \mathtt{w>2})\}$ results in an assertion violation for I.

On the other hand, $\{3, 4\}$ is *not* a minimal $\mathcal{S}_{arb}$-repairable set: For example, the $\mathcal{S}_{arb}$-patch $\{(3, \mathtt{z:=-6})\}$ is an $\mathcal{S}_{arb}$-repair for I. Note that, the $\mathcal{S}_{arb}$-patch only needs to repair the bug, and not the program. That is, it is sufficient that there is no assertion violation on the specific input I, even though an assertion could be violated in the $\mathcal{S}_{arb}$-patched program on another input.

The set of all minimal $\mathcal{S}_{arb}$*-repairable set*s is $\{\{2\}, \{3\}, \{4, 5\}\}$. Therefore, the set of $\mathcal{S}_{arb}$-relevant statements is $\{2, 3, 4, 5\}$. The set of all minimal $\mathcal{S}_{mut}$*-repairable set*s is $\{\{2, 3\}, \{3, 4\}\}$. Therefore, the set of $\mathcal{S}_{mut}$-relevant statements is $\{2, 3, 4\}$.

Fault localization should focus the programmer's attention on locations that are relevant for the bug. But, returning the exact set of $\mathcal{S}$*-relevant* locations, as defined above, can be computationally hard. In practice, what many fault localization algorithms return is a set of locations that *may* be relevant: The returned locations have a higher chance of being $\mathcal{S}$-relevant than those who are not, but there is no guarantee that all returned locations are $\mathcal{S}$-relevant, nor that all $\mathcal{S}$-relevant locations are returned. We call such an algorithm *may fault localization*. In contrast, we define *must fault localization*, as follows:

Definition 1 ($\mathcal{S}$-must location set). *An $\mathcal{S}$-must location set is a set of locations that contains at least one location from each minimal $\mathcal{S}$-repairable set.*[5]

[3] If st is an assignment, its expression is its right-hand-side. If st is a conditional statement, its expression is its condition.

[4] This simple definition of the mutation scheme is used only for simplicity of presentation. Our implementation supports a much richer set of mutation rules, as explained in Sect. 7.

[5] This is, in fact, a hitting set of the set of all minimal $\mathcal{S}$-repairable sets.

Definition 2 ($\mathcal{S}$-must fault localization). *An $\mathcal{S}$-must fault localization algorithm is an algorithm that for every program P and every buggy input I, returns an $\mathcal{S}$-must location set.*

Note that, an $\mathcal{S}$-must location set is not required to contain all $\mathcal{S}$-relevant locations, but only one location from each minimal $\mathcal{S}$-repairable set. Still, this is a powerful notion since it guarantees that no repair is possible without including at least one element from the set.

Also note, that the set of all locations visited by P during its execution on I is always an $\mathcal{S}$-must location set. This is because any $\mathcal{S}$-patch where none of these locations is included is definitely **not** an $\mathcal{S}$-repair, since the same assertion will be violated along the same path. However, this set of locations may not be minimal. In the sequel, we aim at finding small $\mathcal{S}$-must location sets.

Example 2. Continuing the previous example, the set $\{2, 3, 4\}$ is an $\mathcal{S}_{arb}$-must location set, and also an $\mathcal{S}_{mut}$-must location set. In contrast, the set $\{2, 3\}$ is only an $\mathcal{S}_{mut}$-must location set, but not an $\mathcal{S}_{arb}$-must location set, since it does not contain any location from the $\mathcal{S}_{arb}$-minimal repairable set $\{4, 5\}$. The set $\{2\}$ is neither an $\mathcal{S}_{arb}$-must location set nor an $\mathcal{S}_{mut}$-must location set.

Example 3. Consider again the `absValue` procedure of Fig. 1. The set $\{2\}$ is an $\mathcal{S}_{mut}$-minimal repairable set and an $\mathcal{S}_{arb}$-minimal repairable set for the bug in question. Therefore, we can say that all algorithms that were shown in Sect. 2 not to include the location 2 in their result [2,6,14,21,23], are neither $\mathcal{S}_{arb}$-must nor $\mathcal{S}_{mut}$-must fault localization algorithms.

5 Fault Localization Using Program Formula Slicing

In this section we formally define the notion of slicing. Based on this, we present an algorithm for computing must fault localization for $\mathcal{S}_{arb}$ and $\mathcal{S}_{mut}$.

5.1 Program Formula Slicing

A central building block in our fault localization technique is *slicing*. But, we do not define slicing in terms of the program directly, but in terms of the program formula representing it, instead. The input to the slicing algorithm is a program formula φ, a model μ of it, and a variable v. Recall that φ is a conjunction of constraints from S_{assign}, S_{phi} and S_{demand} (see Sect. 3.2). The goal of the slicing algorithm is to compute the *slice* of the variable v with respect to φ and μ. Intuitively, this slice includes the set of all constraints that influence the value v gets in μ.

Similar to traditional slicing, it is easy to define the slice as the reflexive-transitive closure of a dependency relation. But, unlike traditional slicing, which defines dependencies between statements, our dependency relation is between variables of the formula. These variables are indexed. Each originates from a variable of the underlying SSA program, where it was assigned at most once.

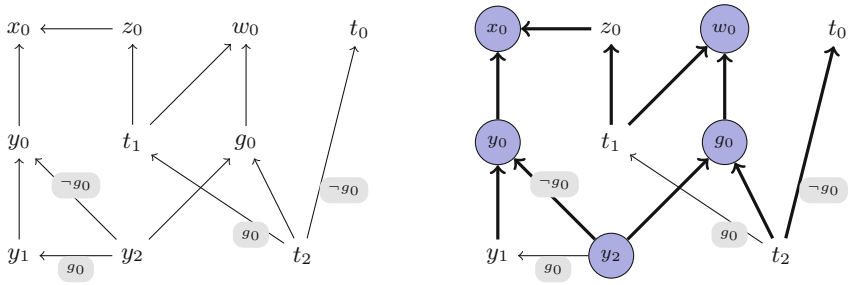


Fig. 3. Illustration of the static and dynamic dependency relations of the `foo procedure`

We refer to variables never assigned as *input variables*, and denote the set containing them by $InputVars$. A variable v that was assigned once is called a *computed variable*, and the (unique) constraint encoding the assignment to it is denoted $Assign(v)$. The set of all computed variables is denoted $ComputedVars$. We also denote by $vars(e)$ the set of variables that appear in a formula or expression e.

Definition 3 (Static Dependency). *The static dependency relation of a program formula φ is $SD_\varphi \subseteq vars(\varphi) \times vars(\varphi)$ s.t.*

$$SD_\varphi = \{(v_1, v_2) \mid \exists e \ s.t. \ (v_1 = e) \in S_{assign}, v_2 \in vars(e)\} \cup$$

$$\{(v, b), (v, v_1), (v, v_2) | \ (v = ite(b, v_1, v_2)) \in S_{phi}\}$$

.

The left-hand-side of Fig. 3 presents the graph for the static dependency relation of the `foo procedure` of Fig. 2. The nodes in the graph are (indexed) variables and there is an arrow from v_1 to v_2 iff $(v_1, v_2) \in SD_\varphi$.

Definition 4 (Dynamic Dependency). *The dynamic dependency relation of a program formula φ and a model μ of φ is $DD_{\varphi,\mu} \subseteq vars(\varphi) \times vars(\varphi)$ s.t.*

$$DD_{\varphi,\mu} = \{(v, v_1) \mid \exists b, v_2 \ s.t. \ (v = ite(b, v_1, v_2)) \in S_{phi}, \ \mu[b] = true\}$$

$$\cup\{(v, v_2) \mid \exists b, v_1 \ s.t. \ (v = ite(b, v_1, v_2)) \in S_{phi}, \ \mu[b] = false\}$$

$$\cup\{(v, b) \mid \exists v_1, v_2 \ s.t. \ (v = ite(b, v_1, v_2)) \in S_{phi}\}$$

$$\cup\{(v, v_1) \mid \exists e \ s.t. \ (v = e) \in S_{assign}, v_1 \in vars(e)\}$$

Note that, dynamic dependency includes only dependencies that coincide with the specific model μ, which determines whether the `then` or the `else` direction

of the `if` is executed. Static dependency, on the other hand, takes both options into account. Thus, $DD_{\varphi,\mu} \subseteq SD_{\varphi}$ for every model μ.

The bold arrows on the right-hand-side of Fig. 3 represent the relation $DD_{\varphi,\mu}$ of the `foo procedure`, for any μ where $\mu[g_0] = false$.

Definition 5 (Influencing Variables). *Given a program formula φ, a model μ of it, and a computed variable v, the set of influencing variables of v with respect to φ and μ is:*

$$InfluenceVars_{\varphi,\mu}(v) = \{v' \mid (v, v') \in (DD_{\varphi,\mu})^*\}$$

The circled nodes on the right-hand-side of Fig. 3 represents the variables that belong to $InfluenceVars_{\varphi,\mu}(y_2)$.

Definition 6 (Program Formula Slice). *Given a program formula φ, a model μ of it, and a computed variable v, the program formula slice of v with respect to φ and μ is:*

$$Slice_{\varphi,\mu}(v) = \{Assign(v') \mid v' \in (InfluenceVars_{\varphi,\mu}(v) \cap ComputedVars)\}$$

Thus, intuitively, $Slice_{\varphi,\mu}(v)$ includes all constraints (in SSA form) encoding assignments that influence the value of v in μ. More precisely, when considering the conjunction of only the constraints of $Slice_{\varphi,\mu}(v)$, as long as the value of all input variables remains the same as in μ, the value of v will remain the same as well. This is formalized in the following theorem, whose proof can be found in the full version [39].

Theorem 2. *For every φ, μ and v, the following holds:*

$$\left[\bigwedge_{c \in Slice_{\varphi,\mu}(v)} c \wedge \bigwedge_{v_i \in InputVars} (v_i = \mu[v_i])\right] \implies (v = \mu[v])$$

Continuing with our example of `foo procedure`,

$$Slice_{\varphi,\mu}(y_2) = \{\ y_2 = ite(g_0, y_1, y_0),\ y_0 = x_0 - 3,\ g_0 = w_0 > 3\}.$$

5.2 Computing the Program Formula Slice

The computation of the program formula slice is composed of two steps. In the first step, we build a graph based on the static dependency relation, SD_{φ}. In the second step, we compute the slice $Slice_{\varphi,\mu}(v)$ by computing the set of nodes reachable from v in this graph, using a customized reachability algorithm, which makes use of the model μ.

The graph built during the first step is called the *Static Dependency Graph (SDG)* of φ. Nodes of this graph are variables of φ and edges are the static dependencies of SD_{φ}. Edges are annotated using the function ψ, mapping every static dependency (v, v') to a boolean formula such that $(v, v') \in DD_{\varphi,\mu}$ iff

$\mu \models \psi[(v, v')]$. Specifically, for every constraint of the form $(v = ite(b, v_1, v_2))$ in S_{phi}, the edge (v, v_1) is annotated with b and the edge (v, v_2) is annotated with $\neg b$. All other edges of the graph are annotated with *true*. See the left-hand-side of Fig. 3. For simplicity all *true* annotations are omitted.

The algorithm for the second step is presented in Algorithm 1. This algorithm gets a program formula φ, its SDG, a model μ of φ, and a variable v, and computes $Slice_{\varphi,\mu}(v)$. First, the set $InfluenceVars_{\varphi,\mu}(v)$ is computed as the set of nodes reachable from v in SDG, except that the reachability algorithm traverses an edge (v, v') only if $\mu \models \psi[(v, v')]$. Thus, an edge (v, v') is traversed iff $(v, v') \in DD_{\varphi,\mu}$, which means that the set of reachable nodes computed this way is in fact $InfluenceVars_{\varphi,\mu}(v)$. Finally, the slice $Slice_{\varphi,\mu}(v)$ is the set of constraints encoding assignments to variables in $InfluenceVars_{\varphi,\mu}(v)$.

Algorithm 1. Compute The Program Formula Slice

Input: a program formula φ, its SDG, a model μ of φ and a variable v.
Output: $Slice_{\varphi,\mu}(v)$.

Procedure $ComputeSlice(\varphi, SDG, \mu, v)$
1: $V := \emptyset$
2: $ModelBasedDFS(SDG, v, \mu, V)$
3: $Slice := \{Assign(v') \mid v' \in V\}$
4: **return** $Slice$

Procedure $ModelBasedDFS(SDG, v, \mu, V)$
1: $V := V \cup \{v\}$
2: **for** $(v, w) \in E$ s.t. $\mu \models \psi[(v, w)]$ **do**
3: **if** $w \notin V$ **then**
4: $ModelBasedDFS(SDG, w, \mu, V)$

Algorithm 2. FOrmula-Slicing-Fault-Localization (FOSFL)

Input: A program formula φ of a program P, and a model μ of φ.
Output: A set of statements F of P.

Procedure $FOSFL(\varphi, \mu)$
1: $SDG := ComputeDependencyGraph(\varphi)$
2: demandFormula := $\bigwedge_{c \in S_{demand}} c$
3: $V := ImportantVars$(demandFormula,μ)
4: $S := \emptyset$
5: **for** $v \in V$ **do**
6: $S := S \cup ComputeSlice(\varphi, SDG, \mu, v)$
7: $F := \emptyset$
8: **for** $c \in S \cap S_{assign}$ **do**
9: $F := F \cup \{Origin(c)\}$
10: **return** F

5.3 The Fault Localization Algorithm

Our fault localization algorithm is presented in Algorithm 2. The input to this algorithm is a program formula φ of a program P, and a model μ of φ. The model μ represents a buggy execution of P on an input I, and the algorithm returns a set of locations, F, that is an $\mathcal{S}_{mut}$-must location set.

As before, we assume to know the origin of constraints in φ, and use the sets S_{assign}, S_{phi} and S_{demand}. Furthermore, here we also assume that for every constraint $c \in S_{assign}$, we know exactly which program statement it came from. We call this statement the *origin* of c, and denote it by $Origin(c)$.

As a first step, the algorithm computes a set of variables V by calling the procedure $ImportantVars$. This procedure receives an SMT formula φ and a model μ of φ, and reduces μ to a partial model of φ. A *partial model* of φ w.r.t. μ is a partial mapping from variables of the formula to values, which is

consistent with μ and is sufficient to satisfy the formula. For example, for the formula $\varphi = (a = 0 \vee b = 0)$ and the model $\mu = \{a \mapsto 0, b \mapsto 1\}$, the valuation $\{a \mapsto 0\}$ is a partial model of φ. Procedure $ImportantVars$ will return the set of variables that appear in the partial model ($\{a\}$ in our example). Details of this procedure are presented in the full version [39].

The formula passed to $ImportantVars$ in our case is the conjunction of all demands in S_{demand}. Recall that the set S_{demand} contains constraints encoding all conditions that need to be met for an assertion violation to happen: Conditions from assumptions appear as is, while conditions from assertions are negated and disjuncted (See Fig. 2. The last constraint on the right-hand-side represents the disjunction of the negated assertions). Therefore, the set of variables V, returned by $ImportantVars$, is such that as long as their values in μ remain the same, this conjunction will still be satisfied, which means that an assertion violation will still occur.

To make sure that their values do *not* remain the same, we use slicing: The algorithm proceeds by computing the program formula slice for each of the variables in V using Algorithm 1. All slices are united into the combined set S. This set represents all constraints that if remain the same, then *all* the variables in V maintain their value. Thus, at least one element from S must be included in any repair.

Note that, by first applying $ImportantVars$, we reduce the number of variables whose value should be preserved in order to maintain the bug. The smaller this number, the smaller F is. We will explain the usefulness of a small F in Sect. 6.

Finally, we need to translate the constraints in S back to statements of P. Because of how the slicing algorithm works, constraints in S may belong to either S_{assign} or S_{phi}. If they belong to S_{phi}, we ignore them, because they encode the control-flow structure of the program, rather than a particular statement. Otherwise, we add the origin of the constraint, which is a statement of the program, to the set of returned locations, F. Note that, several different constraints may have the same origin, for example due to loop unwinding. In such a case, it is sufficient for one constraint encoding the statement st to be included in S, for st to be included in F. A proof for the following theorem can be found in the full version [39].

Theorem 3. *Algorithm FOSFL is an $\mathcal{S}_{arb}$-must and also an $\mathcal{S}_{mut}$-must fault localization algorithm.*

5.4 Incremental Fault Localization

It is often necessary to apply fault localization to several bugs in the same program, or even to several programs with different bugs. Therefore, it is desired that the fault localization algorithm be *incremental*, which means that the computation effort of each fault localization attempt should be proportional to the changes made from the previous attempt. In other words, we should avoid recomputation whenever possible, taking advantage of the fact that the program remains the same, or at least remains similar.

Algorithm FOSFL can be easily made incremental for the case of different bugs of the same program. In this case, several successive calls are made to the algorithm using the same program formula φ, but with different models of it. Since the static dependency relation SD_φ depends solely on the program formula, and not on the model, we can avoid re-computing the SDG for each call. Instead, we can compute the SDG once, upfront, and whenever FOSFL is called, simply skip the first line. We call the incremental version of FOSFL Incremental-Formula-Slicing-Fault-Localization (I-FOSFL).

Note that I-FOSFL is useful not only for fault localization of different bugs of the same program, but also whenever the SDG remains the same during successive fault localization calls. This is the case when considering different mutated programs P' of the same program P, since every change to P' replaces an expression e with an expression e' over the same variables. Thus, the SDG remains the same, since the static dependency relation, in fact, only depends on $vars(e)$, and not on e itself[6].

6 Program Repair with Iterative Fault Localization

In [38], a mutation-based algorithm for program repair, named AllRepair, was presented. This algorithm uses the mutation scheme in order to repair programs with respect to assertions in the code. Unlike fault localization, where the motivation is repairing a bug for a specific input, program repair aims at repairing the program for *all* inputs. To avoid confusion, we refer to a repair for all inputs as a *full repair*. In [38], the notion of a *full repair* is bounded: loops are unwound wb times, and a program is considered *fully repaired* if no assertion is violated along executions with at most wb unwindings. A program that is not fully repaired is said to be *buggy*. For the rest of this section, we refer to an $\mathcal{S}_{mut}$-patch as a patch, and to an $\mathcal{S}_{mut}$-patched program as a mutated program.

As its name implies, the goal of AllRepair is to obtain all *minimal* fully repaired mutated programs, where minimality refers to the patch used in the program. It goes through an iterative generate-validate process. The generate phase chooses a mutated program from the search space, and the validate phase checks whether this program is fully repaired, by solving its program formula. The mutated program is fully repaired iff the formula is unsatisfiable.

The generate-validate process is realized using an interplay between a SAT solver and an SMT solver. The SAT solver is used for the generate stage. For every mutation M and line l, there is a boolean variable $B_M(l)$, which is true if and only if mutation M is applied to line l. A boolean formula is constructed and sent to the SAT solver, where each satisfying assignment corresponds to a program in the search space. The SMT solver is used for the validate stage. The program formula of the mutated program is solved to check if it is buggy

[6] This is true for $\mathcal{S}_{mut}$ but not for $\mathcal{S}_{arb}$, since the latter allows to replace an expression e with an expression e' over different variables.

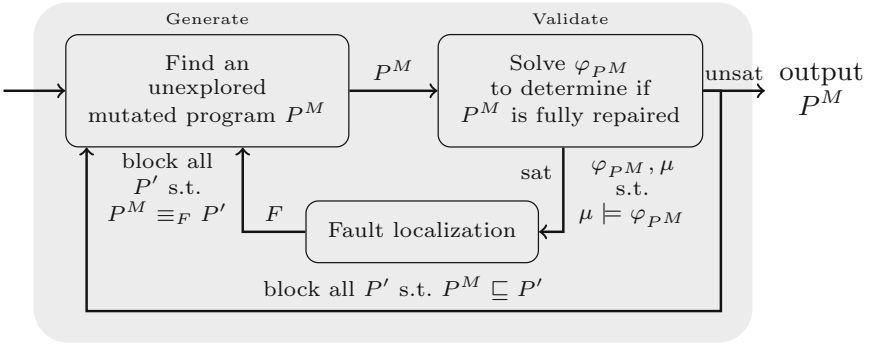


Fig. 4. Algorithm FL-AllRepair: Mutation-based program repair with iterative fault localization. The notation $P^M \equiv_F P'$ means that P^M and P' agree on the content of all locations in F. The notation $P^M \sqsubseteq P'$ means that the patch used for creating P' is a superset of the patch used for creating P^M.

or not. To achieve minimality, when a mutated program created using a patch τ is fully repaired, every mutated program created using a patch τ', with $\tau \subseteq \tau'$, is blocked.

Example 4. Let P^M be a fully repaired mutated program obtained by applying the patch τ, consisting of mutating line l_1 using mutation M_1 and mutating line l_2 using mutation M_2. Then blocking any superset of τ will we done by adding to the boolean formula representing the search space, the blocking clause $\neg(B_{M_1}(l_1) \wedge B_{M_2}(l_2))$, which means "either do not apply M_1 to l_1 or do not apply M_2 to l_2". This clause blocks any mutated program with $\tau \subseteq \tau'$.

Blocking such programs prunes the search space, but only in a limited way. No pruning occurs when the mutated program is buggy.

In this paper, we extend the algorithm of [38] with a fault localization component. The goal of the new component is to prune the search space by identifying sets of mutated programs that are buggy, without inspecting each of the individual programs in the set.

Figure 4 shows the program repair algorithm with the addition of fault localization. In the new algorithm, called FL-AllRepair, whenever a mutated program is found to be buggy during the validation step, its program formula is passed to the fault localization component along with the model obtained when solving the formula. The fault localization component returns a set of locations F, following the I-FOSFL algorithm. Since this set is guaranteed to be an $\mathcal{S}_{mut}$-must location set, at least one of the locations in it should be changed for the bug to be fixed. Consequently, all mutated programs in which all locations from F remain unchanged are blocked from being explored in the future. As before, blocking is done by adding a blocking clause that disallows such programs.

Example 5. Let P^M be a buggy mutated program for which F consists of $\{l_1, l_2, l_3\}$, where l_1 was mutated with M_1, l_2 was not mutated, and l_3 was mutated with M_3. The blocking clause $\neg B_{M_1}(l_1) \vee \neg B_{Original}(l_2) \vee \neg B_{M_3}(l_3)$ will be added

to the boolean formula representing the search space of mutated programs. It restricts the search space to those mutated programs that either do not apply mutation M_1 to l_1, or do mutate l_2 or do not apply M_3 to l_3. This will prune from the search space all mutated programs which are identical to P^M on the locations in F. Note that smaller F will result in a larger set of pruned programs.

Proposition 1. *Algorithm* FL-AllRepair *is sound and complete.*

7 Experimental Results

We have implemented our fault localization technique and its integration with mutated-based program repair in the tool AllRepair, available at https://github.com/batchenRothenberg/AllRepair. In this section, we present experiments evaluating the contribution of the new fault localization component to the program repair algorithm. We refer to the algorithm of [38], without fault localization, as AllRepair, and to the algorithm presented in this paper as FL-AllRepair. Both algorithms search for minimal *wb*-violation free programs, and both are sound and complete. Thus, for every buggy program and every bound *wb*, both algorithms will eventually produce the same list of repairs.

The difference between the algorithms lies in the repair loop. In case a mutated program is found to be buggy, the AllRepair algorithm will only block the one program, while the FL-AllRepair algorithm might block a set of programs. Therefore, the number of repair iterations required to cover the search space can only decrease using the FL-AllRepair algorithm. On the other hand, the cost of each iteration with fault localization is strictly higher than without it. Our goal in this evaluation is to check if the use of fault localization pays off. That is, to check if repairs are produced faster using FL-AllRepair than using AllRepair.

Benchmarks. For our evaluation, we have used programs from two benchmarks: TCAS and Codeflaws. The TCAS benchmark is part of the Siemens suite [12], and is frequently used for program repair evaluation [5,34,38]. The TCAS program implements a traffic collision avoidance system for aircrafts, and consists of approximately 180 lines of code. We have used all 41 faulty versions of the benchmark in our experiments.

The Codeflaws benchmark [41] is also a well-known and widely used benchmark for program repair. Programs in this benchmark are taken from buggy user submissions to the programming contest site Codeforces[7]. In each program, a user tries to solve a programming problem published as part of a contest on the site. The programming problems are varied, and also the users have a diverse level of expertise. The benchmark also provides correct versions for all buggy versions, which are used to classify bug types by computing the syntactic difference. For our experiments we randomly chose 13 buggy versions classified with bug types that can be fixed using mutations. The size of the chosen programs ranges from 17 to 44 lines of code.

[7] http://codeforces.com/.

Mutations. The mutations used in AllRepair (and accordingly in FL-AllRepair) is a subset of the mutations used in [37]. We define two *mutation levels*, where level 1 contains only a subset of the mutations available in level 2. Thus, level 1 involves easier computation but may fail more often in finding repairs.

Table 1 shows the list of mutations used in each mutation level. For example, for the category of arithmetic operator replacement, in mutation level 1, the table specifies two sets: $\{+,-\}$ and $\{/,\%\}$. This means that a $+$ can be replaced by a $-$, and vice versa, and that the operators $/$, $\%$ can be replaced with each other. Constant manipulation mutations apply to a numeric constant and include increasing its value by 1 ($C \rightarrow C\ +1$), decreasing it by 1 ($C \rightarrow C\ -1$), setting it to 0 ($C \rightarrow 0$) and changing its sign ($C \rightarrow -C$).

Level 1	Level 2
$\{+,-\}$, $\{/,\%\}$	$\{+,-,*\}$, $\{/,\%\}$
$\{>,>=\}$, $\{<,<=\}$	$\{>,>=,<,<=\}$, $\{==,!=\}$
$\{\|\|, \&\ \&\ \}$	
$\{>>,<<\},\{\&, \|, \hat{}\ \}$	
	$C \rightarrow C\ +1$, $C \rightarrow C\ -1$, $C \rightarrow -C$, $C \rightarrow 0$

Table 1. Partition of mutations to levels

Setting. All of our experiments were run on a Linux 64-bit Ubuntu 16.0.4 virtual machine with 1 CPU, 4 GB of RAM and 40 GB of storage, provided using the VMware vRA service[8]. For each of the buggy versions in our benchmarks we have experimented with both mutation levels 1 and 2. For the Codeflaws benchmarks we additionally experimented with different unwinding bounds: 2 (entering the loop once), 5, 8 and 10. This experiment is irrelevant to the TCAS benchmarks since the TCAS program does not contain loops or recursive calls. Overall we had 186 combinations of buggy programs, mutation levels and unwinding bounds. We refer to each such combination as an *input*. For each input, we run both the AllRepair and the FL-AllRepair algorithms with a timeout of 10 minutes and a mutation size limit of 2 (i.e., at most two mutations could be applied at once).

7.1 Results

In total, 131 different repairs were found during our experiments, for 60 different inputs (for several inputs there was more than one possible repair). In this count, we treat repairs fixing the same program in the same way as different, if they were produced using different mutation levels or unwinding bounds. This is because our evaluation is concerned with the time to find these repairs, and both the mutation level and the unwinding bound greatly influence this time.

Because the time to produce a repair sometimes varied in several orders of magnitude depending on the input, we have chosen to split repairs into three categories: fast, intermediate, and slow, and examine the time difference separately for each category. Splitting repairs to categories was done according to the time it took to find them using the AllRepair algorithm. If that time was

[8] https://www.vmware.com/il/products/vrealize-automation.html.

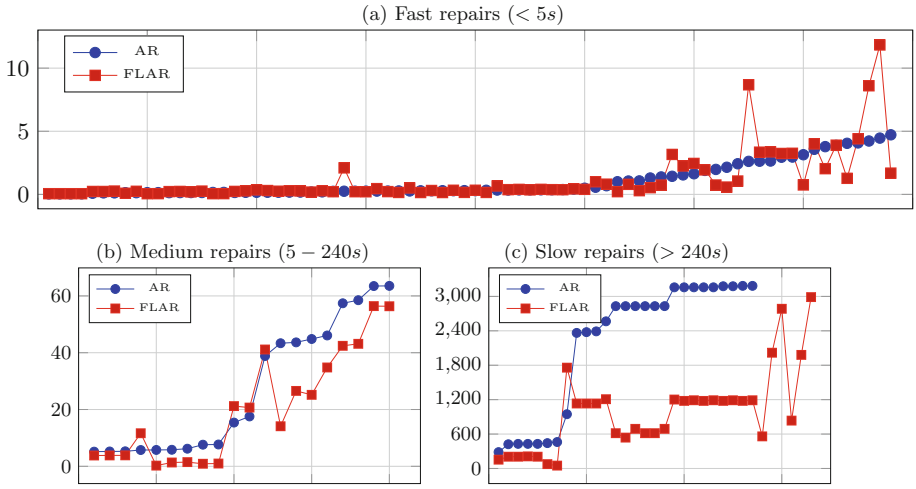


Fig. 5. Time to find each repair using AllRepair (AR) and FL-AllRepair (FLAR). Each x value represents a single repair, and the corresponding y values represent the time, in seconds, it took to find that repair using both algorithms. Note that the graphs differ in the y axis scale.

under 5 seconds, the repair was considered fast. If it was over 4 minutes, it was considered slow, and otherwise it was considered intermediate.

Figure 5 shows a comparison of the time, in seconds, it took to find repairs in both algorithms. There are three graphs, according to our three categories. In all graphs, each x value represents a single repair, where the corresponding blue dot in the y axis represents the time it took to find that repair using AllRepair, and the red square represents the time using FL-AllRepair. So, whenever the blue dot is above the red square, FL-AllRepair was faster in finding that repair, and the y difference represents the time saved.

For the fast category (Fig. 5a), there is no clear advantage to FL-AllRepair. The majority of the repairs in this category are produced in less than a second using both algorithms. For the remaining repairs, there appears to be as many cases where FL-AllRepair is faster as when it is slower. But, in all cases where there is a time difference, in either direction, it is only of a few seconds.

For the intermediate category (Fig. 5b), the advantage of FL-AllRepair is starting to become clear. There are now only 4 repairs (out of 20) for which FL-AllRepair is slower. Also, on average, it is slower by 4 seconds, but faster by 10 seconds. Finally, for the slow category (Fig. 5c), there is an obvious advantage to FL-AllRepair. First, it is able to find 6 repairs *exclusively*, while AllRepair reaches a time-out. Also, for the remaining 27 repairs, FL-AllRepair is faster in all cases but one. The time difference is now also very significant: FL-AllRepair is faster by 1512 seconds (around 25 minutes) on average.

To sum up, the results show that in many cases our algorithm FL-AllRepair is able to save time in finding repairs. The savings are especially significant in

cases where it takes a long time to produce the repair using the original AllRepair algorithm, and these are the cases where time savings are most needed.

7.2 Comparison with Other Repair Methods

The TCAS benchmark was recently used also in [34], where AllRepair's performance was compared to that of four other automated repair tools: Angelix [29], GenProg [26], FoRenSiC [5] and Maple [34]. AllRepair was found to be faster by an order of magnitude than all of the compared tools, taking only 16.9 seconds to find a repair on average, where the other tools take 1540.7, 325.4, 360.1, and 155.3 seconds, respectively. Since in our experiments on TCAS fl-AllRepair was faster than AllRepair on average (and even when it was slower it was only by a few seconds), we conclude that fl-AllRepair also compares favorably to these other tools.

In terms of repairability, the repair scheme used by AllRepair (and fl-AllRepair) is limited compared to the other tools: AllRepair only uses mutations on expressions while Angelix, FoRenSiC and Maple allow replacing an expression with a template (e.g., a linear combination of variables), which is then filled out to create a repair. GenProg allows modifying a statement as well as deleting it or adding a statement after it. Therefore, the other tools are inherently capable of producing repairs in more cases than AllRepair.

In the case of TCAS, the study showed that AllRepair is able to find repairs for 18 versions (a result that we confirm in our experiments as well), while Angelix, GenProg, FoRenSiC and Maple found 32, 11, 23 and 26, respectively. But, what the study also showed, is that in repair methods that are based on tests, in many cases the repair found only adhered to the test-suite, but was not correct when inspected manually. When counting only correct repairs, AllRepair finds repairs for 18 versions (all of AllRepairs repairs are correct), while Angelix, GenProg, FoRenSiC and Maple find 9, 0, 15 and 26, respectively. Since fl-AllRepair is able to find all repairs found by AllRepair, the same results also apply to fl-AllRepair.

8 Related Work

Dynamic slicing has been widely used for fault localization in the past [16,36, 43,45–47]. But, as we have seen, traditional notations of dynamic slicing [2,23] are not must (with respect to neither of the presented schemes), and thus, the above techniques may fail to include relevant locations in their results.

Other approaches for fault localization include spectrum-based (SBFL) [1, 13,20,31,44], mutation-based (MBFL) [15,18,30,35] and formula-based (FBFL) [7,14,17,21,40]. Both SBFL and MBFL techniques compute the suspiciousness of a statement using coverage information from failing and passing test executions. MBFL uses, in addition, information on how test results change after applying different mutations to the program. Both SBFL and MBFL techniques can be seen as may fault localization techniques, in nature: they return locations that

are likely to be relevant to the failing execution, based on all executions. We see may fault localization techniques as orthogonal to ours (and to must fault localization techniques in general), since in the trade-off between returning a small set of locations, and returning one that is guaranteed to contain all relevant statements, may techniques prefer the first, while must techniques prefer the second. In the context of repair, there are interesting applications for both.

FBFL techniques represent an error trace using an SMT formula and analyze it to find suspicious locations. These techniques include using error invariants [6,14,17,40], maximum satisfiability [21,24,25], and weakest preconditions [7]. What we were able to show in this paper, is that the methods of [6,14,21] are not must. In contrast, we believe (though we do not prove it) that the methods of [7,24,25] are must. But, what [7,24,25] have in common is that they use the semantics of the error trace or the program. Though semantic information can help to further minimize the number of suspicious locations, retrieving it involves using expensive solving-based procedures. Our approach, on the other hand, uses only syntactic information, which makes the fault localization computation relatively cheap; No SMT solving is needed. Thus, these approaches can be seen as complementary to ours.

In the literature there is also a wide range of techniques for automated program repair using formal methods [4,10,19,22,29,32,33,42]. Both [11] and [37] also use fault localization followed by applying mutations for repair. But, unlike this work, fault localization is applied only for the original program. Also, neither the Tarantula fault localization used in [11] nor the dynamic slicing used in [37] carries the guarantee of being a must fault localization. The tool MUT-APR [3] fixes binary operator faults in C programs, but only targets faults that require one line modification. The tools FoREnSiC [5] and Maple [34] repair C programs with respect to a formal specification, but they do so by replacing expressions with templates, which are then patched and analysed. SemGraft [28] conducts repair with respect to a reference implementation, but relies on tests for SBFL fault localization of the original program.

9 Conclusion

In this work we define a novel notion of *must* fault localization, that carefully identifies program locations that are relevant for a bug, so that the set is sufficiently small but is guaranteed not to miss desired repairs. We also show that the notion of *must* fault localization should be defined with respect to the repair scheme in use. We show that our notion of must fault localization is particularly useful in pruning the search space of a specific mutation-based repair algorithm.

To the best of our knowledge, we are the first to investigate the widely-used notion of fault localization and to suggest criteria for evaluating its different implementation.

References

1. Abreu, R., Zoeteweij, P., Van Gemund, A.J.C.: An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006, pp. 39–46 (2006)
2. Agrawal, H., Horgan, J.R.: Dynamic Program Slicing. In: PLDI, pp. 246–256 (1990)
3. Assiri, F.Y., Bieman, J.M.: MUT-APR: MUTation-based automated program repair research tool. In: Arai, K., Kapoor, S., Bhatia, R. (eds.) FICC 2018. AISC, vol. 887, pp. 256–270. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-03405-4_17
4. Attie, P.C., Dak, K., Bab, A.L., Sakr, M.: Model and program repair via SAT solving. ACM Trans. Embed. Comput. Syst. **17**(2), 1–25 (2017)
5. Bloem, R., Drechsler, R., Fey, G., Finder, A., Hofferek, G., Könighofer, R., Raik, J., Repinski, U., Sülflow, A.: FoREnSiC– an automatic debugging environment for C programs. In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 260–265. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_24
6. Christ, J., Ermis, E., Schäf, M., Wies, T.: Flow-sensitive fault localization. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 189–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_13
7. Christakis, M., Heizmann, M., Mansur, M.N., Schilling, C., Wüstholz, V.: Semantic fault localization and suspiciousness ranking. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 226–243. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_13
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. Clarke, E., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of the Design Automation Conference, 2003, pp. 368–371. IEEE (2003)
10. D'Antoni, L., Samanta, R., Singh, R.: Qlose: program repair with quantitative objectives. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 383–401. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_21
11. Debroy, V., Wong, W.E.: Using mutation to automatically suggest fixes for faulty programs. In: 2010 Third International Conference on Software Testing, Verification and Validation (ICST), pp. 65–74. IEEE (2010)
12. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir. Softw. Eng. **10**(4), 405–435 (2005)
13. Eric Wong, W., Debroy, V., Choi, B.: A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw. **83**(2), 188–208 (2010)
14. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 187–201. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_17
15. Gong, P., Zhao, R., Li, Z.: Faster mutation-based fault localization with a novel mutation execution strategy. In: Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2015, pp. 1–10. IEEE (2015)

16. Hofer, B., Wotawa, F.: Spectrum enhanced dynamic slicing for better fault localization. ECAI **242**, 420–425 (2012)
17. Holzer, A., Schwartz-Narbonne, D., Tabaei Befrouei, M., Weissenbacher, G., Wies, T.: Error invariants for concurrent traces. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 370–387. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_23
18. Hong, S., Lee, B., Kwak, T., Jeon, Y., Ko, B., Kim, Y., Kim, M.: Mutation-based fault localization for real-world multilingual programs. In: ASE, pp. 464–475 (2015)
19. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_23
20. Jones, J., Harrold, M., Stasko, J.: Visualization for fault localization. In: Proceedings of ICSE 2001 Workshop on Software Visualization, pp. 71–75 (2001)
21. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI, pp. 437–446 (2011)
22. Kneuss, E., Koukoutos, M., Kuncak, V.: Deductive program repair. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 217–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_13
23. Korel, B., Laski, J.: Dynamic program slicing. Inf. Process. Lett. **29**, 155–163 (1988)
24. Lamraoui, S.-M., Nakajima, S.: A formula-based approach for automatic fault localization of multi-fault programs. J. Inf. Process. **24**, 88–98 (2016)
25. Lamraoui, S.-M., Nakajima, S., Hosobe, H.: Hardened flow-sensitive trace formula for fault localization. In: ICECCS (2015)
26. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
27. Liu, K., Koyuncu, A., Bissyande, T.F., Kim, D., Klein, J., Le Traon, Y.: You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In: ICST, pp. 102–113 (2019)
28. Mechtaev, S., Nguyen, M.-D., Noller, Y., Grunske, L., Roychoudhury, A.: Semantic program repair using a reference implementation. In: ICSE (2018)
29. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: ICSE (2016)
30. Moon, S., Kim, Y., Kim, M., Yoo, S.: Ask the mutants: mutating faulty programs for fault localization. In: ICST (2014)
31. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. **20**(3), 1–32 (2011)
32. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 772–781. IEEE Press (2013)
33. Nguyen, T.V., Weimer, W., Kapur, D., Forrest, S.: Connecting program synthesis and reachability: automatic program repair using test-input generation. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 301–318. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_17
34. Nguyen, T.-T., Ta, Q.-T., Chin, W.-N.: Automatic program repair using formal verification and expression templates. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 70–91. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_4
35. Papadakis, M., Traon, Y.L.: Metallaxis-FL: mutation-based fault localization. Softw. Test. Verif. Reliab. **21**(3), 195–214 (2015)

36. Qian, J., Xu, B.: Scenario oriented program slicing. In: Proceedings of the ACM Symposium on Applied Computing, pp. 748–7752 (2008)
37. Repinski, U., Hantson, H., Jenihhin, M., Raik, J., Ubar, R., Guglielmo, G.D., Pravadelli, G., Fummi, F.: Combining dynamic slicing and mutation operators for ESL correction. In: 2012 17th IEEE European Test Symposium (ETS), pp. 1–6. IEEE (2012)
38. Rothenberg, B.-C., Grumberg, O.: Sound and complete mutation-based program repair. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 593–611. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_36
39. Rothenberg, B.-C., Grumberg, O.: Must fault localization for program repair. https://batg.cswp.cs.technion.ac.il/wp-content/uploads/sites/78/2020/05/MustFaultLocalizationForProgramRepairCav2020.pdf, May 2020. A full version of the CAV 2020 paper of the same title
40. Schäf, M., Schwartz-Narbonne, D., Wies, T.: Explaining inconsistent code. In: ESEC/FSE, pp. 521–531 (2013)
41. Tan, S.H., Yi, J., Yulis, Mechtaev, S., Roychoudhury, A.: Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In: Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017, pp. 180–182 (2017)
42. von Essen, C., Jobstmann, B.: Program repair without regret. Form. Methods Syst. Des. **47**(1), 26–50 (2015). https://doi.org/10.1007/s10703-015-0223-6
43. Wang, Y., Patil, H., Pereira, C., Lueck, G., Gupta, R., Neamtiu, I.: DrDebug: deterministic replay based cyclic debugging with dynamic slicing. In: Proceedings of the 12th ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2014, pp. 98–108 (2014)
44. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Trans. Reliab. **63**(1), 290–308 (2014)
45. Wotawa, F.: Fault localization based on dynamic slicing and hitting-set computation. In: Proceedings of the International Conference on Quality Software, pp. 161–170 (2010)
46. Zhang, X., Gupta, N., Gupta, R.: A study of effectiveness of dynamic slicing in locating real faults. Empir. Softw. Eng. **12**(2), 143–160 (2007)
47. Zhang, X., Gupta, N., Gupta, R.: Locating faulty code by multiple points slicing. Softw. Pract. Exp. **39**(7), 661–699 (2007)

Author Index