12. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: Creignou, N., Le Berre, D. (eds.) Theory and Applications of Satisfiability Testing – SAT 2016. pp. 228–245. Springer International Publishing, Cham (2016)
13. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Hardware and Software: Verification and Testing. pp. 50–65. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
14. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020), https://doi.org/10.1007/s10817-019-09525-z
15. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**(5), 506–521 (1999), https://doi.org/10.1109/12.769433
16. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001), https://doi.org/10.1145/378239.379017
17. Nötzli, A., Barbosa, H., Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Reconstructing fine-grained proofs of rewrites using a domain-specific language. In: Griggio, A., Rungta, N. (eds.) Formal Methods in Computer-Aided Design - 22nd Conference, FMCAD 2022, Trento, Italy, October 17-21, 2022, Proceedings. pp. 65–74. Formal Methods in Computer-Aided Design, TU Wien Academic Press (2022)
18. Rollini, S.F., Bruttomesso, R., Sharygina, N., Tsitovich, A.: Resolution proof transformation for compression and interpolation. Formal Methods Syst. Des. **45**(1), 1–41 (2014), https://doi.org/10.1007/s10703-014-0208-x
19. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in CakeML. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 223–241. Springer (2021), https://doi.org/10.1007/978-3-030-72013-1_12
20. Van Gelder, A.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008 (2008), http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf
21. Vyskocil, J., Stanovský, D., Urban, J.: Automated proof compression by invention of new definitions. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 447–462. Springer (2010), https://doi.org/10.1007/978-3-642-17511-4_25
22. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 8561, pp. 422–429 (2014)

Unsatisfiability Proofs for Distributed Clause-Sharing SAT Solvers

Dawn Michaelson[2(✉)], Dominik Schreiber[3(✉)], Marijn J. H. Heule[1,4], Benjamin Kiesl-Reiter[1], and Michael W. Whalen[1,2]

[1] Amazon Web Services, Seattle, USA
[2] University of Minnesota, Minneapolis, USA
micha576@umn.edu
[3] Karlsruhe Institute of Technology, Karlsruhe, Germany
dominik.schreiber@kit.edu
[4] Carnegie Mellon University, Pittsburgh, USA

Abstract. Distributed clause-sharing SAT solvers can solve problems up to one hundred times faster than sequential SAT solvers by sharing derived information among multiple sequential solvers working on the same problem. Unlike sequential solvers, however, distributed solvers have not been able to produce proofs of unsatisfiability in a scalable manner, which has limited their use in critical applications. In this paper, we present a method to produce unsatisfiability proofs for distributed SAT solvers by combining the partial proofs produced by each sequential solver into a single, linear proof. Our approach is more scalable and general than previous explorations for parallel clause-sharing solvers, allowing use on distributed solvers without shared memory. We propose a simple sequential algorithm as well as a fully distributed algorithm for proof composition. Our empirical evaluation shows that for large-scale distributed solvers (100 nodes of 16 cores each), our distributed approach allows reliable proof composition and checking with reasonable overhead. We analyze the overhead and discuss how and where future efforts may further improve performance.

Keywords: SAT solving · proofs · distributed computing.

1 Introduction

SAT solvers are general-purpose tools for solving complex computational problems. By encoding domain problems into propositional logic, users have successfully applied SAT solvers in various fields such as formal verification [31], automated planning [25], and mathematics [8, 16]. The list of applications has grown significantly over the years, mainly because algorithmic improvements have led to orders of magnitude improvement in the performance of the best sequential solvers (see, e.g., [21] for a comparison).

Despite all this progress, there are still many problems that cannot be solved quickly with even the best sequential solvers, pushing researchers to explore ways of parallelizing SAT solving. One approach that has worked well for specific problem instances is *Cube-and-Conquer* [17, 18], which can achieve near-linear

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 348–366, 2023.
https://doi.org/10.1007/978-3-031-30823-9_18

speedups for thousands of cores but requires domain knowledge about how effectively to split a problem into subproblems. An alternative approach that does not require such knowledge is *clause-sharing portfolio solving*, which has recently led to solvers [12,28] achieving impressive speedups (10x–100x on a 100x16 core cluster) over the best sequential solvers across broad sets of benchmarks.[5]

Although distributed solvers are demonstrably the most powerful tools for solving hard SAT problems, there is an important caveat: unlike sequential solvers, current distributed clause-sharing solvers cannot produce proofs of unsatisfiability. While there has been foundational work in producing proofs for shared-memory clause-sharing SAT solvers [14], existing approaches are neither scalable nor general enough for large-scale distributed solvers. This is not just a theoretical problem—for four problems in the 2020 and 2021 SAT competitions, distributed solvers produced incorrect answers that were not discovered until the 2022 competition because they could not be independently verified.[6]

In this paper, we deal with this issue and present the first scalable approach for generating proofs for distributed SAT solvers. To construct proofs, we maintain *provenance* information about shared clauses in order to track how they are used in the global solving process, and we use the recently-developed LRAT proof format [9] to track dependencies among partial proofs produced by solver instances. By exploiting these dependencies, we are then able to reconstruct a single linear proof from all the partial proofs produced by the sequential solvers. We first present a simple sequential algorithm for proof reconstruction before devising a parallel algorithm that can even be implemented in a distributed way. Both algorithms produce independently-verifiable proofs in the LRAT format. We demonstrate our approaches using an LRAT-producing version of the sequential SAT solver CaDiCaL [5] to turn it into a clause-sharing solver, and then modify the distributed solver Mallob [28] to orchestrate a portfolio of such CaDiCaL instances while tracking the IDs of all shared clauses.

We conduct an evaluation of our approaches from the perspective of efficiency, benchmarking the performance of our clause-sharing portfolio solver against the winners of the cloud track, parallel track, and sequential track from the SAT Competition 2022. Adding proof support introduces several kinds of overhead for clause-sharing portfolios in terms of solving, proof reconstruction, and proof checking, which we examine in detail. We show that even with this overhead, distributed solving and proving is much faster than the best sequential approaches. We also demonstrate that our approach dramatically outperforms previous work on proof production for clause-sharing portfolios [14]. We argue that much of the overhead of our current setup can be compensated, among other measures, by improving support for LRAT in solver backends. We thus hope that our work provides an impetus for researchers to add LRAT support to other solvers.

Our main contributions are as follows:

[5] c.f.: the SAT Competition 2022 results: https://satcompetition.github.io/2022/downloads/sc2022-detailed-results.zip

[6] The incorrectly scored problems were `SAT_MS_sat_nurikabe_p08.pddl_71.cnf`, `randomG-Mix-n18-d05.cnf`, `php12e12.cnf`, and `Cake_9_20.cnf`.

- We present the first effective and scalable approach for proof generation in distributed SAT solving.
- We implement our approach on top of the state-of-the-art solvers CaDiCaL and Mallob.
- We perform a large-scale empirical evaluation analyzing the overhead introduced by proof production as compared to state-of-the-art portfolios.
- We demonstrate that our approach dramatically outperforms previous work in parallel proof production, and that it remains substantially more scalable than the best sequential solvers.

The rest of this paper is structured as follows. In Section 2, we present the background required to understand the rest of our paper and discuss related work. In Section 3, we describe the general problem of producing proofs for distributed SAT solving and a simple algorithm for proof combination. In Section 4, we describe a much more efficient distributed version of our algorithm before discussing implementation details in Section 5. Finally, we present the results of our empirical evaluation in Section 6 and conclude with a summary and an outlook for future work in Section 7.

2 Background and Related Work

The Boolean satisfiability problem (SAT) asks whether a Boolean formula can be satisfied by some assignment of truth values to its variables. An overview can be found in [6]. We consider formulas in *conjunctive normal form* (CNF). As such, a formula F is a conjunction (logical "AND") of disjunctions (logical "OR") of literals, where a literal is a Boolean variable or its negation. For example, $(\overline{a} \vee b \vee c) \wedge (b \vee \overline{c}) \wedge (a)$ is a formula with variables a, b, c and three clauses. A *truth assignment* $\mathcal{A}$ maps each variable to a Boolean value (true or false). A formula F is *satisfied* by an assignment $\mathcal{A}$ if F evaluates to true under $\mathcal{A}$, and F is *satisfiable* if such an assignment exists. Otherwise, F is called *unsatisfiable*.

If a formula F is found to be satisfiable, modern SAT solvers commonly output a truth assignment; users can easily evaluate F under the assignment in linear time to verify that F is indeed satisfiable. In contrast, if a formula turns out unsatisfiable, sequential SAT solvers produce an independently-checkable proof that there exists no assignment that satisfies the formula.

File Formats in Practical SAT Solving. In practical SAT solving, formulas are specified in the DIMACS format. DIMACS files feature a header of the form '`p cnf #variables #clauses`' followed by a list of clauses, one clause per line. For example, the clause $(x_1 \vee \overline{x}_2 \vee x_3)$ is represented as '`1 -2 3 0`'. An example formula in DIMACS format is given in Figure 1.

The current standard format for proofs is DRAT [15]. DRAT files are similar to DIMACS files, with each line containing a proof statement that is either an *addition* or a *deletion*. Additions are lines that represent clauses like in the DIMACS format; they identify clauses that were derived ("learned") by the solver. Each clause addition must preserve satisfiability by adhering to the so-called

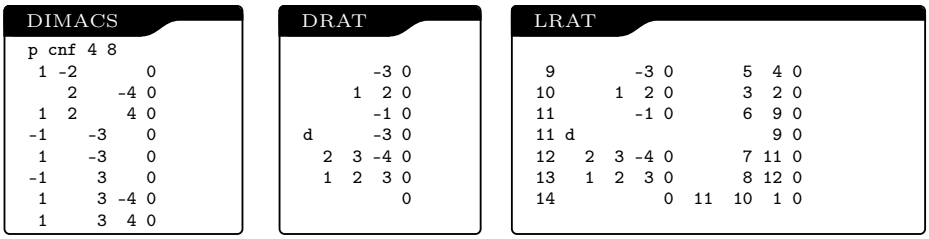


Fig. 1: DIMACS formula and corresponding proofs in DRAT and LRAT format.

RAT criterion—as the details of RAT are not essential to our paper, we refer the reader to the respective literature for more details [20]. Deletions are lines that start with a '`d`', followed by a clause; they identify clauses that were deleted by the solver because they were not deemed necessary anymore. Clause deletions can only make a formula "more satisfiable", meaning that they aren't required for deriving unsatisfiability, but they drastically speed up proof checking. A valid DRAT proof of unsatisfiability ends with the derivation of the empty clause. As the empty clause is trivially unsatisfiable (and since each proof step preserves satisfiability) the unsatisfiability of the original formula can then be concluded. An example DRAT proof is given in Figure 1.

The more recent LRAT proof format [9] augments each clause-addition step with so-called *hints*, which identify the clauses that were required to derive the current clause. This makes proof checking more efficient, and in fact the usual pipeline for trusted proof checking is to first use an efficient but unverified tool (like DRAT-trim [15]) to transform a DRAT proof into an LRAT proof, and then check the resulting LRAT proof with a formally verified proof checker (c.f., [9, 13, 22, 30]). Figure 1 shows an LRAT proof corresponding to a DRAT proof. Each proof line starts with a clause ID. The numbering starts with `9` because the eight clauses of the original formula are assigned the IDs `1` to `8`. Each clause addition first lists the literals of the clause, then a terminating `0`, followed by hints (in the form of clause IDs), and finally another `0`. For example, clause `9` contains the literal `-3` and can be derived from the clauses `4` and `5` of the original formula. Clause deletions just state the clause ID of the clause that is to be deleted, as in the later deletion of clause `9`. In our work, we exploit the hints of LRAT to determine dependencies among distributed solvers.

Parallel and Distributed SAT Solving. One way to parallelize SAT solving is to run a portfolio of sequential solvers in parallel and to consider a problem solved as soon as one of the solvers finishes (c.f. [1, 4, 5, 11, 12, 18, 23, 29, 32]). Given that the solvers are sufficiently diverse, portfolio solving is already effective if all of the sequential solvers work independently, but performance and scalability can be boosted significantly by having the solvers share information in the form of learned clauses [4, 12]. This approach is taken by the distributed solver Mallob [28], which won the cloud track of the last three SAT competitions [2, 3, 27]. As opposed to other solvers, Mallob relies on a communication-efficient aggrega-

tion strategy to collect the globally most useful learned clauses and to reliably filter duplicates as well as previously shared clauses [27]. With this strategy, which aims to maximize the density and utility of the communicated data, Mallob scored first place in all four eligible subtracks for unsatisfiable problems at the 2022 SAT Competition.

As we discuss in more detail later, the drawback of clause sharing is that a local proof written by an individual solver may contain clauses whose derivations cannot be justified because they rely on clauses imported from another solver. Previous work focuses on writing DRAT proofs for clause-sharing parallel solvers [14]. In that work, solvers write to the same shared proof as they learn clauses. However, since the clauses are shared, one solver deleting a clause could invalidate a later clause-addition by another solver that is still holding the clause. To handle this, the parallel solver moderates deletion statements, only writing them to the proof once all solvers have deleted a clause, which leads to poor scalability during proof search. In our approach, solvers write proof files fully independently—only when the unsatisfiability of the problem has been determined do we combine all proofs into a single valid proof.

Other recent work includes reconstructing proofs from divide-and-conquer solvers [24] and from a particular shared-memory parallel solver [10] whereas we aim to exploit distributed portfolio solving.

3 Basic Proof Production

Our goal is to produce checkable unsatisfiability proofs for problems solved by distributed clause-sharing SAT solvers. We propose to reuse the work done on proofs for sequential solvers by having each solver produce a partial proof containing the clauses it learned. These partial proofs are invalid in general because each sequential solver can rely on clauses shared by other solvers when learning new clauses. For example, when solver A derives a new clause, it might rely on clauses from solvers B and C, which in turn relied on clauses from solvers D and E, and so on. The justification of A's clause derivation is thus spread across multiple partial proofs. We need to combine the partial proofs into a single valid proof in which the clauses are in *dependency order*, meaning that each clause can be derived from previous clauses.

To generate an efficiently-checkable combined proof in a scalable way, we must solve three challenges:

1. Provide metadata to identify which solver produced each learned clause.
2. Efficiently sort learned clauses in dependency order across all solvers.
3. Reduce proof size by removing unnecessary clauses.

Switching from DRAT to the LRAT proof format provides the mechanism to unlock all three challenges. First, we specialize the clause-numbering scheme used by LRAT in order to distinguish the clauses produced by each solver. Second, we use the dependency information from LRAT to construct a complete proof from the partial proofs produced by each solver. Finally, we determine which clauses are unnecessary (or used only for certain parts of the proof) to delete clauses from the proof as soon as they are no longer required.

Algorithm 1 Algorithm for combining partial proofs

```
1: function COMBINE(partial proofs p_1, p_2, ...p_n, number of original clauses o)
2:     i ← 1
3:     while true do
4:         if p_i.hasNext() then
5:             ⟨id, type, clause, proofHint⟩ ← p_i.peekNext()
6:             if dependenciesSatisfied(proofHint) then
7:                 emit ⟨id, type, clause, proofHint⟩
8:                 p_i.next()                                   ▷ Line completed
9:                 if clause = ∅ then                           ▷ Derived empty clause
10:                    return
11:            else                          ▷ Leave the line and move to next partial proof
12:                i ← (i mod n) + 1
13:        else                              ▷ Move to next partial proof if current is done
14:            i ← (i mod n) + 1
```

We update the clause-distribution mechanism in the distributed solver to broadcast the clause ID with each learned clause. A receiving solver stores the clause with its ID and uses the ID in proof hints when the clause is used locally, as it does with locally-derived clauses. Unlike locally-derived clauses, we add no derivation lines for remote clauses to the local proof. Instead, these derivations will be added to the final proof when combining the partial proofs.

3.1 Solver Partial Proof Production

To combine the partial proofs into a complete proof, we modify the mechanism producing LRAT proofs in each of the component solvers. We assign to each clause an ID that is unique across solvers and identifies which solver originally derived it. The following mapping from clauses to IDs achieves this:

Definition 1. *Let o be the number of clauses in the original formula and let n be the number of sequential solvers. Then, the ID of the k-th derived clause ($k \geq 0$) of solver i is defined as $ID_k^i = o + i + nk$.*

Given ID_k^i, we can easily determine the solver ID i using modular arithmetic.

3.2 Partial Proof Combination

Once the distributed solver has concluded the input formula is unsatisfiable, we have n partial proofs. The clause derivations in these proofs refer to clauses of other partial proofs, but they are, locally, in dependency order. We can therefore combine the partial proofs without reordering their clauses beforehand. We can simply interleave their clauses so the resulting proof is also in dependency order, ignoring any deletions in the partial proofs.

Our algorithm goes through the partial proofs round-robin, at each step emitting all the clauses from each file where the dependencies of the clause have

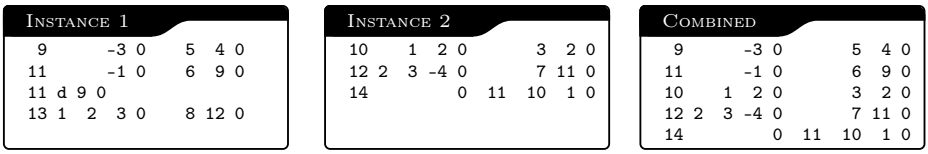


Fig. 2: Partial proofs and combined proof of unsatisfiability.

already been emitted. It ends when the empty clause is emitted. The procedure is shown in Algorithm 1. For each partial proof, we maintain an iterator over the learned clauses. We add the next clause from the current partial proof (p_i) to the final proof if its dependencies are satisfied (determined by comparing each hint to the last clause emitted from the partial proof whence it originated); otherwise it cycles to the next partial proof. It emits the line and moves to the next clause in the file. The algorithm terminates when it emits the empty clause (line 10).

Example 1. Suppose that two solver instances (instance 1 and instance 2) determined together that the formula from Figure 1 is unsatisfiable, with the two partial proofs shown in Figure 2. We start with instance 1. As clause 9 only relies on original clauses, we emit it. Clause 11 relies on original clause 6 and emitted clause 9, so we emit it. Clause 13 relies on clauses 8 and 12, which is not emitted, so we cannot emit clause 13 and move to instance 2. Clause 10 can be emitted, as can clause 12, which relies on an original and an emitted clause. Clause 14 relies on emitted clauses 11 and 10 and on original clause 1, so we can emit it as well. Since clause 14 is the empty clause, we finish with a complete proof, shown in Figure 2(c). Notice that clause 13 was not added to the combined proof, since it was not required to satisfy any dependencies of the empty clause.

3.3 Proof Pruning

The combined proof produced by our procedure is valid but not efficiently checkable because (1) it can contain clauses that are not required to derive the empty clause and (2) it does not contain deletion lines, meaning that a proof checker must maintain *all* learned clauses in memory throughout the checking process. To reduce size and to improve proof-checking performance, we prune our combined proof toward a minimal proof containing only necessary clauses, and we add deletion statements for clauses as soon as they are not needed anymore.

Algorithm 2 shows our pruning algorithm that walks the combined proof in reverse (similar to backward checking of DRAT proofs [19]). We maintain a set of clauses *required* in the proof, initialized to the empty clause alone. We then process all clauses in reverse order, including the empty clause, ignoring all clauses not in the required set. For each required clause, we check its dependencies to see if this is the first time (from the proof's end) a dependency is seen; if so, we emit a deletion line for the dependency since it will never be used again in the proof. After checking all its dependencies, we output the clause itself. The

Algorithm 2 Algorithm for pruning proofs

```
1: function PRUNE(combined and reversed proof p, number of original clauses o)
2:     required ← {p.peekNextId()}        ▷ Must be empty clause, which is required
3:     while p.hasNext() do
4:         ⟨id, type, clause, proofHint⟩ ← p.readNext()
5:         if id ∈ required then          ▷ Only process a line if it is required later
6:             for hint ∈ proofHint do
7:                 if hint > o ∧ hint ∉ required then          ▷ Not used later
8:                     required ← required ∪ {hint}
9:                     emit ⟨id, delete, hint⟩
10:            emit ⟨id, add, clause, proofHint⟩
```

final output of the algorithm is a proof in reversed order, where each clause is required for some derivation and deleted as soon as it is no longer required.

Example 2. Consider the combined proof from Figure 2. After applying Algorithm 2, working backward from clause 14, we determine that clause 12 is not required, so it is removed. Additionally, prior to clause 11, clause 9 is not in the required set, so it can be deleted after processing clause 11. On larger proofs, as discussed in Section 6, pruning can reduce the size of the proof by 10x or more.

4 Distributed Proof Production

The proof production as described above is sequential and may process huge amounts of data, all of which needs to be accessible from the machine that executes the procedure. In addition, maintaining the required clause IDs during the procedure may require a prohibitive amount of memory for large proofs. In the following, we propose an efficient distributed approach to proof production.

4.1 Overview

Our previous sequential proof-combination algorithm first combines all partial proofs into a single proof and then prunes unneeded proof lines. In contrast, our distributed algorithm first prunes all partial proofs in parallel and only then merges them into a single file.

We have m processes with c solver instances each, amounting to a total of $n = mc$ solvers. We make use of the fact that the solvers exchange clauses in periodic intervals (one second by default). We refer to these intervals between subsequent sharing operations as *epochs*. Consider Fig. 3 (left): Clause 118 was produced by S_2 in epoch 1. Its derivation may depend on local clause 114 and on any of the 11 clauses produced in epoch 0, but it cannot depend, e.g., on clause 109 or 111 since these clauses have been produced after the last clause sharing. More generally, a clause c produced by instance i during epoch e can only depend on (i) earlier clauses by instance i produced during epoch e or earlier, and (ii) clauses by instances $j \neq i$ produced *before* epoch e.

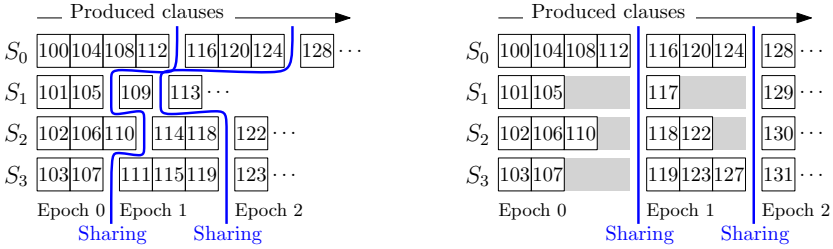


Fig. 3: Four solvers work on a formula with 99 original clauses, produce new clauses (depicted by their ID), and share clauses periodically, without (left) and with (right) aligning clause IDs.

Using this knowledge, we can essentially *rewind* the solving procedure. Each process reads its partial proofs in reverse order, outputs each line which adds a required clause, and adds the hints of each such clause to the required clauses. Required remote clauses produced in epoch e are transferred to their process of origin before any proof lines from epoch e are read. As such, whenever a process reads a proof line, it knows whether the clause is required. The outputs of all processes can be merged into a single valid proof (Section 4.3).

4.2 Distributed Pruning

Clause ID Alignment. To synchronize the reading and redistribution of clause IDs in our distributed pruning, we need a way to decide from which epoch a remote clause ID originates. However, solvers generally produce clauses with different speeds, so the IDs by different solvers will likely be in dissimilar ranges within the same epoch over time. For instance, in Fig. 3 (left) instance S_3 has no way of knowing from which epoch clause 118 originates. To solve this issue, we propose to align all produced clause IDs after each sharing. During the solving procedure, we add a certain offset δ_i^e to each ID produced by instance i in epoch e. As such, we can associate each epoch e with a global interval $[A_e, A_{e+1})$ that contains all clause IDs produced in that epoch. In Fig. 3 (right), $A_0 = 100$, $A_1 = 116$, and $A_2 = 128$. Clause 118 on the left has been aligned to 122 on the right ($\delta_2^1 = 4$) and due to $A_1 \leq 122 < A_2$ all instances know that this clause originates from epoch 1.

Initially, $\delta_i^0 := 0$ for all i. Let I_i^e be the first original (unaligned) ID produced by instance i in epoch e. With the sharing that initiates epoch $e > 0$, we compute the common start of epoch e, $A_e := \max_i\{I_i^e + \delta_i^{e-1} - i\}$, as the lowest possible value that is larger than all clause IDs from epoch $e-1$. We then compute offsets δ_i^e in such a way that $I_i^e + \delta_i^e = A_e + i$, which yields $\delta_i^e := (A_e + i) - I_i^e$. If we then export a clause produced during e by instance i, we add δ_i^e to its ID, and if we import shared clauses to i, we filter any clauses produced by i itself. Note that we do not modify the solvers' internal ID counters or the proofs they output. Later, when reading the partial proof of solver i at epoch e, we need to add δ_i^e to each ID originating from i. All other clause IDs are already aligned.

Rewinding the Solve Procedure. Assume that instance $u \in \{1, \ldots, n\}$ has derived the empty clause in epoch $\hat{e}$. For each local solver i, each process has a *frontier* F_i of required clauses produced by i. In addition, each process has a *backlog* B of remote required clauses. B and F_i are collections of clause IDs and can be thought of as maximum-first priority queues. Initially, F_u contains the ID of the empty clause while all other frontiers and backlogs are empty. Iteration $x \geq 0$ of our algorithm processes epoch $\hat{e} - x$ and features two stages:

1. Processing: Each process continues to read its partial proofs in reverse order from the last introduced clause of the current epoch. If a line from solver i is read whose clause ID is at the top of F_i, then the ID is removed from F_i, the line is output, and each clause ID hint h in the line is treated as follows:

- h is inserted in F_j if local solver j (possibly $j = i$) produced h.
- h is inserted in B if a remote solver produced h.
- h is dropped if h is an ID of an original clause of the problem.

Reading stops as soon as a line's ID precedes epoch $e = \hat{e} - x$. Each F_i as well as B now only contain clauses produced *before* e.

2. Task redistribution: Each process extracts all clause IDs from B that were produced during $\hat{e} - x - 1$. These clause IDs are aggregated among all processes, eliminating duplicates in the same manner as Mallob's clause sharing detects duplicate clauses [28]. Each process traverses the aggregated clause IDs, and each clause produced by a local solver i is added to F_i.

Our algorithm stops in iteration $\hat{e}$ after the Processing stage, at which point all frontiers and backlogs are empty and all relevant proof lines have been output.

Analysis. In terms of total work performed, all partial proofs are read completely. For each required clause we may perform an insertion into some B, a deletion from said B, an insertion into some F_i, and a deletion from said F_i. If we assume logarithmic work for each insertion and deletion, the work for these operations is linear in the combined size of all partial proofs and loglinear in the size of the output proof. In addition, we have $\hat{e}$ iterations of communication whose overall volume is bounded by the communication done during solving. In fact, since only a subset of shared clauses are required and we only share 64 bits per clause, we expect strictly less communication than during solving. Computing A_e for each epoch e during solving is negligible since the necessary aggregation and broadcast can be integrated into an existing collective operation. Regarding memory usage, the size of each B and each F_i can be proportional to the combined size of all required lines of the according partial proofs. However, we can make use of external data structures which keep their content on disk except for a few buffers.

4.3 Merging Step

For each partial proof processed during the pruning step, we have a stream of proof lines sorted in reverse chronological order, i.e., starting with the highest clause ID. The remaining task is to merge all these lines into a single, sorted proof file. As shown in Fig. 4 (left), we arrange all processes in a tree. We can easily merge a number of sorted input streams into a single sorted output stream

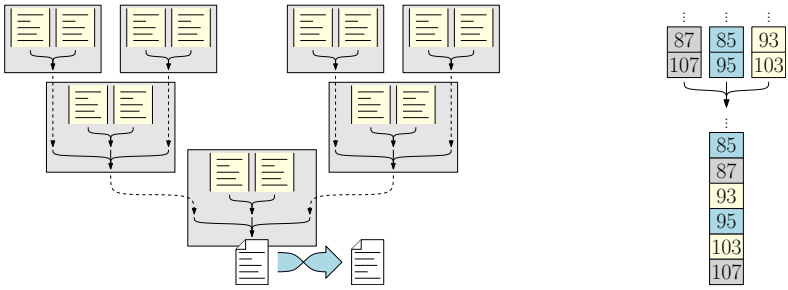


Fig. 4: Left: Proof merging with seven processes and 14 solvers. Each box represents a process with two local proof sources. Dashed arrows denote communication. Right: Example of merging three streams of LRAT lines into a single stream. Each number i represents an LRAT line describing a clause of ID i.

by repeatedly outputting the line with the highest ID among all inputs (Fig. 4 right). This way, we can hierarchically merge all streams along the tree. At the tree's root, the output stream is directed into a file. This is a sequential I/O task that limits the speed of merging. Finally, since the produced file is in reverse order, a buffered operation reverses the file's content.

A final challenge is to add clause deletions to the final proof. Before a line is written to the combined proof file, we can scan its hints and output a deletion line for each hint we did not encounter before (see Section 3.3). However, implementing this in an exact manner requires maintaining a set of clause IDs which scales with the final proof size. Since our proof remains valid even if we omit some clause deletions, we can use an approximate membership query (AMQ) structure with fixed size and a small false positive rate, e.g., a Bloom filter [7].

5 Implementation

We employ a solver portfolio based on the sequential SAT solver CaDiCaL [5]. We modified CaDiCaL to output LRAT proof lines and to assign clause IDs as described in Section 3.1. To ensure sound LRAT proof logging, some features of CaDiCaL currently need to be turned off, such as bounded variable elimination, hyper-ternary resolution, and vivification. Similarly, Mallob's original portfolio of CaDiCaL configurations features several options that are incompatible with our proof logging as of yet. Therefore, we created a smaller portfolio of "safe" configurations that include shuffling variable priorities, adjusted restart intervals, and disabled inprocessing. We also use different random seeds and use Mallob's diversification based on randomized initial variable polarities.

We modified Mallob to associate each clause with a 64-bit clause ID. For consistent bookkeeping of sharing epochs, we defer clause sharing until all processes have fully initialized their solvers. While several solvers may derive the empty clause simultaneously, only one of them is selected to be the "winner" whose empty clause will be traced. The distributed proof production features

communication similar to Mallob's clause sharing. To realize the frontier F_i and the backlog B described in Section 4.2, we implemented an external-memory data structure which writes clause IDs to disk, categorized by their epoch. Upon reaching a new epoch, all clause IDs from this epoch are read from disk and inserted into an internal priority queue to allow for efficient polling and insertion. To merge the pruned partial proofs, we use point-to-point messages to query and send buffers of proof lines between processes. We interleave this merging with the pruning procedure in order to avoid writing the intermediate output to disk. We use a fixed-size Bloom filter to add some deletion lines to the final proof.

6 Evaluation

In this section, we present an evaluation of our proof production approaches. We provide the associated software as well as a digital appendix online.[7]

6.1 Experimental Setup

Supporting proofs introduces several kinds of performance overhead for clause-sharing portfolios in terms of solving, proof reconstruction, and proof checking. We wish to examine how well our proof-producing solver performs against (1) best-of-breed parallel and cloud solvers that do not produce proofs, (2) previous approaches to proof-producing parallel solvers, and (3) best-of-breed sequential solvers. We analyze the overhead introduced by each phase of the process, and we discuss how and where future efforts might improve performance.

We use the following pipeline for our proof-producing solvers: First, the input formula is preprocessed by performing exhaustive unit propagation. This is necessary due to a technical limitation of our LRAT-producing modification of CaDiCaL. Second, we execute our proof-producing variant of Mallob on the preprocessed formula. Third, we prune and combine all partial proofs, using either our sequential proof production or our distributed proof production. Fourth, we merge the preprocessor's proof and our produced proof and syntactically transform the result to bring the set of clause IDs into compact shape. Fifth and finally, we run `lrat-check`[8] to check the final proof. Only steps two and three of our pipeline are parallelized (step three depending on the particular experiment). We will refer to the first two steps as *solving*, the third step as *assembly*, the fourth step as *postprocessing*, and the fifth step as *checking*.

To examine performance overhead for proof-producing parallel and distributed solvers, we compare our proof-producing cloud and parallel solvers (`mallob-cacld-p` and `mallob-capar-p`) against six solvers. First, we include the winners of the 2022 SAT competition cloud track (`mallob-kicaliglu`, using Kissat+CaDiCaL+Lingeling+Glucose), parallel track (`parkissat-rs`, using Kissat), and sequential track (`Kissat_MAB-HyWalk`), as well as the second place

[7] https://github.com/domschrei/mallob/tree/certified-unsat
[8] https://github.com/marijnheule/drat-trim

Table 1: Overview of solved instances: (S)equential, (P)arallel, and (C)loud

Solver	Type	Solved	SAT	UNSAT	PAR-2 score
`Kissat_MAB-HyWalk`	S	218	118	100	1065.7
`parkissat-rs`	P	299	155	144	603.0
`mallob-ki`	P	260	113	147	827.6
`mallob-capar`	P	292	145	147	641.6
`mallob-capar-p` (Seq.)	P	279	140	139	719.8
`mallob-capar-p` (Par.)	P	276	141	135	731.4
`mallob-kicaliglu`	C	341	165	176	344.8
`mallob-cacld`	C	333	163	170	378.0
`mallob-cacld-p`	C	314	159	155	484.1

solver from the parallel track (`mallob-ki`, using Lingeling[9]). We then run a parallel and cloud version of Mallob that runs our described CaDiCaL portfolio *without* proof production (`mallob-capar` and `mallob-cacld`).

Following the SAT competition setup, each cloud solver runs on 100 m6i.4xlarge EC2 instances (16 core, 64GB RAM), each parallel solver runs on a single m6i.16xlarge EC2 instance (64 core, 256GB RAM), and the sequential `Kissat_MAB-HyWalk` runs on a single m6i.4xlarge EC2 instance. For each solver, we run the full benchmark suite from the SAT-Competition 2022 (400 formulas) containing both SAT and UNSAT examples. The timeout for the solving step is 1000 seconds, and the timeout for all subsequent steps is set to 4000 seconds.

Since earlier work [14] is no longer competitive in terms of solving time, we only compare proof-checking times. Specifically, we measure the overhead of checking un-pruned DRAT proofs as the ones produced by [14]. As such, we can get a picture of the performance of the earlier approach if it was realized with state-of-the-art solving techniques. We generate un-pruned DRAT proofs from the original (un-pruned) LRAT proof by stripping out the dependency information and adding delete lines for the last use of each clause.

6.2 Results

First we examine the performance overhead of changing portfolios to enable proof generation as described in Section 5 on the *solving process only*. Fig. 5 (left) and Table 1 show this data. The PAR-2 metric takes the average time to solve each problem, but counts a timeout result as a 2x penalty (e.g., given our timeout of 1000 seconds, a timeout is scored as taking 2000 seconds). We can see that our CaDiCaL portfolio `mallob-capar` outperforms the Lingeling-based `mallob-ki` significantly and is almost on par with `parkissat-rs`. Similarly, `mallob-cacld` solves eight instances less compared to `mallob-kicaliglu` but performs almost equally well otherwise. In both cases, we have constructed solvers which are,

[9] `mallob-ki` employed a Lingeling-based portfolio due to a misconfiguration, see: http://algo2.iti.kit.edu/schreiber/downloads/mallob-ki-mallob-li.pdf

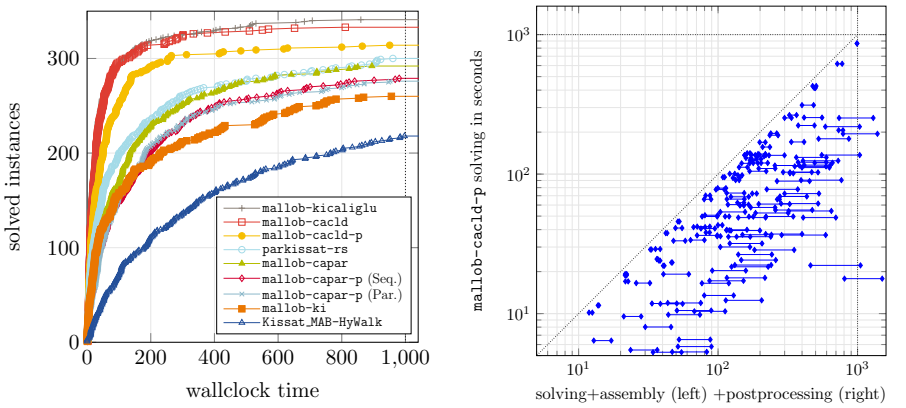


Fig. 5: Left: Comparison of solving times. Right: Relation of solving times to assembly and postprocessing times for `mallob-cacld-p`. Each pair of points corresponds to one instance, the y coordinate denoting the solving time. The left x coordinate denotes solving and assembly time and the right x coordinate denotes solving, assembly, and postprocessing time.

up to a small margin, on par with the state of the art. For our actual proof-producing solvers, `mallob-capar-p` and `mallob-cacld-p`, we noticed a more pronounced decline in solving performance. On top of the overhead introduced by proof logging and our preprocessing, we experienced a few technical problems, including memory issues[10], which resulted in a drop in the number of instances solved and also caused `mallob-capar-p` with parallel proof production to solve three instances less than with sequential proof production. We believe that we can overcome these issues in future versions of our system. That being said, our proof-producing solvers already outperform any of the solvers at a lower scale.

Second, we examine statistics on proof reconstruction and checking, showing results in Table 2. Since we want to investigate our approaches' overhead compared to pure solving, we measure run times as a multiple of the solving time. (We provide absolute run times in the Appendix, Table 1.) The prefix "Seq." denotes `mallob-capar-p` with sequential proof production, "Par." denotes `mallob-capar-p` with distributed proof production run on a single machine, and "Cld." denotes `mallob-cacld-p` with distributed proof production.

DRAT checking succeeded in 81 out of 139 cases and timed out in 58 cases. For the successful cases, DRAT checking took 24.8× the solving time on average whereas our sequential assembly, postprocessing and checking combined succeeded in 139 cases and only took 3.8× the solving time on average. This result confirms that our approach successfully overcomes the major scalability problems of earlier work [14]. In terms of uncompressed proof sizes, our LRAT

[10] We disabled Mallob's memory panic mode to ensure consistent proof logging.

Table 2: Statistics on proof production and checking. All properties except for file sizes and pruning factor are given as a multiple of the solving time. We list minima, maxima, medians, arithmetic means, and the 10th and 90th percentiles.

Property	#	min	p10	med	mean	p90	max
DRAT check	81	0.512	1.725	7.442	24.815	67.065	169.869
Seq. assembly	139	0.019	0.305	1.376	2.324	5.747	13.289
Seq. postprocessing	139	0.001	0.012	0.131	0.263	0.790	2.218
Seq. checking	139	0.007	0.043	0.572	1.252	3.970	10.980
Seq. asm+post+chk	139	0.037	0.412	2.110	3.840	10.834	26.487
Par. assembly	135	0.059	0.080	0.365	0.805	2.227	7.475
Par. postprocessing	135	0.001	0.016	0.156	0.293	0.861	2.300
Par. checking	135	0.007	0.042	0.622	1.241	3.540	11.645
Par. asm+post+chk	135	0.067	0.167	1.097	2.339	6.611	21.420
Cld. assembly	155	0.114	0.185	1.412	2.444	5.410	44.268
Cld. postprocessing	155	0.003	0.060	0.696	2.046	4.785	39.096
Cld. checking	155	0.033	0.189	3.291	8.883	21.974	170.378
Cld. asm+post+chk	155	0.168	0.577	5.110	13.373	32.484	253.742
DRAT proof size (GiB)	139	0.012	0.366	1.236	3.246	8.395	29.308
Seq. proof size (GiB)	139	0.016	0.223	2.379	5.384	16.082	46.986
Par. proof size (GiB)	135	0.006	0.173	2.034	5.345	13.164	57.739
Cld. proof size (GiB)	155	0.016	0.342	3.940	10.533	30.130	89.106
Cld. pruning factor	155	2.374	5.379	17.826	293.762	337.486	12466.700

proofs can be about twice as large as the DRAT proofs, which seems more than acceptable considering the dramatic difference in performance. Given that DRAT-based checking was ineffective at the scale of parallel solvers, we decided to omit it in our distributed experiments which feature even larger proofs.

Regarding `mallob-capar-p` with parallel proof production, we can see that the assembly time is reduced from 2.32× down to 0.81× the solving time on average, which also improves overall performance (3.84× to 2.34×).

The results for `mallob-cacld-p` demonstrate that our proof assembly is feasible, taking around 2.5× the solving time on average. We visualized this overhead and how it relates to the postprocessing overhead in Fig. 5 (right). The proofs produced are about twice as large as for `mallob-capar-p`. Considering that the proofs originate from 25 times as many solvers, this increase in size is quite modest, which can be explained by our proof pruning. We captured the *pruning factor* — the number of clauses in all partial proofs divided by the number of clauses in the combined proof — for each instance. Our pruning reduces the derived clauses by a factor of 293.8 on average (17.8 for the median instance), showing that it is a crucial technique to obtain proofs that are feasible to check. As such, we also managed to produce and check a proof of unsatisfiability for a formula whose unsatisfiability has not been verified before (`PancakeVsInsertSort_8_7.cnf`).

Lastly, to compare our approach at the largest scale with the state of the art in sequential solving, we computed speedups of `mallob-cacld-p`, solv-

ing times only, over `Kissat_MAB-HyWalk` and arrived at a median speedup of 11.5 (Appendix, Table 2). We also analyzed `drat-trim` checking times of `Kissat_MAB-HyWalk`, kindly provided by the competition organizers, and arrived at a median overhead of 1.1× its own solving time (Appendix, Table 3). Going by these measures, `Kissat_MAB-HyWalk` takes around $11.5 \cdot 2.1 \approx 24.2\times$ the solving time of `mallob-cacld-p` to arrive at a checked result while our complete pipeline only takes 5.1× the solving time for the median instance. This indicates that our approach is considerably faster than the best available sequential solvers.

We can see that the bottleneck of our pipeline shifts from the assembly step further to the postprocessing and checking steps when increasing the degree of parallelism. This is to be expected since the latter steps are, so far, inherently sequential whereas our proof assembly is scalable. While the postprocessing step is a technical necessity in our current setup, we believe that large portions of it can be eliminated in the future with further engineering. For instance, enhancing the LRAT support of our modified CaDiCaL to natively handle unit clauses in the input would allow us to skip preprocessing and simplify postprocessing.

7 Conclusion and Future Work

Distributed clause-sharing solvers are currently the fastest tools for solving a wide range of difficult SAT problems. Nevertheless, they have previously not supported proof-generation techniques, leading to potential soundness concerns. In this paper, we have examined mechanisms to add efficient support for proof generation to clause-sharing portfolio solvers. Our results demonstrate that we can, with reasonable efficiency, add support to these solvers to have full confidence that the results they produce are correct.

Following our research, more work is required to reduce overhead in the different steps involved and to improve scalability of the end-to-end procedure. This may include designing more efficient (perhaps even parallel) LRAT checkers, examining proof-streaming techniques to eliminate most I/O operations, and improving LRAT support in solver backends. In fact, it might be possible to generalize our approach to DRAT-based solvers by adding additional metadata, and this might allow easier retrofitting of the approach onto larger portfolios of solvers. We also intend to investigate producing proofs in Mallob for the case where many problems are solved at once and jobs are rescaled dynamically [26].

Acknowledgments

We would like to thank Mario Carneiro for providing help for his FRAT-supporting variant of CaDiCaL; Markus Iser for providing competition data on proof checking; Andrew Gacek for his suggestions to early drafts of this paper; and the reviewers for their helpful feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agr. No. 882500). This project was partially supported by the U.S. National Science Foundation grant CCF-2015445.

References

1. Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: Sinz, C., Egly, U. (eds.) Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8561, pp. 197–205. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_15
2. Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions. Department of Computer Science Report Series B, Department of Computer Science, University of Helsinki, Finland (2021)
3. Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions. Department of Computer Science Report Series B, Department of Computer Science, University of Helsinki, Finland (2020)
4. Balyo, T., Sanders, P., Sinz, C.: HordeSat: A massively parallel portfolio SAT solver. In: Heule, M., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2015. pp. 156–172. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_12
5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009), http://dblp.uni-trier.de/db/series/faia/faia185.html
7. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970). https://doi.org/10.1145/362686.362692
8. Brakensiek, J., Heule, M., Mackey, J., Narváez, D.E.: The resolution of Keller's conjecture. J. Autom. Reason. **66**(3), 277–300 (2022). https://doi.org/10.1007/s10817-022-09623-5
9. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 220–236. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_14
10. Fleury, M., Biere, A.: Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses. In: Pragmatics of SAT (2022). https://doi.org/10.48550/arXiv.2207.13577
11. Gomes, C.P., Selman, B.: Algorithm portfolios. Artif. Intell. **126**(1-2), 43–62 (2001). https://doi.org/10.1016/S0004-3702(00)00081-3
12. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. J. Satisf. Boolean Model. Comput. **6**(4), 245–262 (2009). https://doi.org/10.3233/sat190070
13. Heule, M., Jr., W.A.H., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September

26-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10499, pp. 269–284. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_18
14. Heule, M., Manthey, N., Philipp, T.: Validating unsatisfiability results of clause sharing parallel sat solvers. In: POS@ SAT. pp. 12–25 (2014)
15. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **abs/1610.06229** (2016). https://doi.org/10.48550/arXiv.1610.06229
16. Heule, M.J.H.: Schur number five. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18). pp. 6598–6606. AAAI Press (2018). https://doi.org/10.1609/aaai.v32i1.12209
17. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 228–245. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_15
18. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Hardware and Software: Verification and Testing. pp. 50–65. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34188-5_8
19. Heule, M.J., Hunt, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: 2013 Formal Methods in Computer-Aided Design. pp. 181–188 (2013). https://doi.org/10.1109/FMCAD.2013.6679408
20. Järvisalo, M., Heule, M.J., Biere, A.: Inprocessing rules. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7364, pp. 355–370. Springer (2012). https://doi.org/10.1007/978-3-642-31365-3_28
21. Kissat SAT solver. http://fmv.jku.at/kissat/, accessed: 2022-08-17
22. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2020). https://doi.org/10.1007/s10817-019-09525-z
23. Le Frioux, L., Baarir, S., Sopena, J., Kordon, F.: Painless: A framework for parallel SAT solving. In: Gaspers, S., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing – SAT 2017. pp. 233–250. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66263-3_15
24. Nair, A., Chattopadhyay, S., Wu, H., Ozdemir, A., Barrett, C.: Proof-stitch: Proof combination for divide and conquer SAT solvers. In: Formal Methods in Computer-Aided Design. pp. 84–88 (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2
25. Rintanen, J.: Planning and SAT. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 483–504. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-483
26. Sanders, P., Schreiber, D.: Decentralized online scheduling of malleable NP-hard jobs. In: European Conference on Parallel Processing. pp. 119–135. Springer (2022). https://doi.org/10.1007/978-3-031-12597-3_8

27. Schreiber, D.: Mallob in the SAT competition 2022. In: Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions. pp. 46–47. Department of Computer Science Report Series B, University of Helsinki (2022)
28. Schreiber, D., Sanders, P.: Scalable SAT solving in the cloud. In: Li, C.M., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing – SAT 2021. pp. 518–534. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-80223-3_35
29. Schubert, T., Lewis, M., Becker, B.: Pamiraxt: Parallel SAT solving with threads and message passing. J. Satisf. Boolean Model. Comput. **6**(4), 203–222 (2009). https://doi.org/10.3233/sat190068
30. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in cakeml. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 223–241. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_12
31. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proc. IEEE **103**(11), 2021–2035 (2015). https://doi.org/10.1109/JPROC.2015.2455034
32. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008). https://doi.org/10.1613/jair.2490

Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format*

Bruno Andreotti[1], Hanna Lachnitt[2], Haniel Barbosa[1(✉)]

[1] Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
hbarbosa@dcc.ufmg.br
[2] Stanford University, Stanford, USA

Abstract. Proofs from SMT solvers ensure correctness independently from implementation, which is often a requirement when solvers are used in safety-critical applications or proof assistants. Alethe is an established SMT proof format generated by the solvers veriT and cvc5, with reconstruction support in the proof assistants Isabelle/HOL and Coq. The format is close to SMT-LIB and allows both coarse- and fine-grained steps, facilitating proof production. However, it lacks a stand-alone checker, which harms its usability and hinders its adoption. Moreover, the coarse-grained steps can be too expensive to check and lead to verification failures. We present Carcara, an independent proof checker and elaborator for Alethe, implemented in Rust. It aims to increase the adoption of the format by providing push-button proof-checking for Alethe proofs, focusing on efficiency and usability; and by providing elaboration for coarse-grained steps into fine-grained ones, increasing the potential success rate of checking Alethe proofs in performance-critical validators, such as proof assistants. We evaluate Carcara over a large set of Alethe proofs generated from SMT-LIB problems and show that it has good performance and its elaboration techniques can make proofs easier to check.

1 Introduction

Satisfiability modulo theories (SMT) solvers are widely used as background tools in various formal method applications, ranging from proof assistants to program verification [9]. Since these applications rely on the SMT solver results, they must trust their correctness. However, state-of-the-art SMT solvers are often found to have bugs, despite the best efforts of developers [30, 38]. One way to address this issue is to formally verify the solvers' correctness ("certifying" them), but this approach can be prohibitively expensive and time consuming, besides often requiring performance compromises [19, 20, 27, 33] and increasing the evolution cost of the systems [14]. Alternatively, solvers can produce proofs: independently checkable certificates that justify the correctness of their results. Since proof checking generally has lower complexity than solving, small and trusted checkers can verify solver results in an scalable manner. Despite the successful adoption

* This work was partially supported by an Amazon Research Award (Spring 2021), a gift from Amazon Web Services, and the Stanford Center for Automated Reasoning.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 367–386, 2023.
https://doi.org/10.1007/978-3-031-30823-1_19

of this approach by several SMT solvers [7,13,15,24,37], no standard SMT proof format has emerged, with each system using their own format and independent toolchain. The Alethe[1] format [35] for SMT proofs however can be emitted by the veriT solver for several years [10] and recently[2] also by the cvc5 solver [7]. Moreover, Alethe proofs can be reconstructed within the proof assistants Coq [4, 16] and Isabelle/HOL [11,36], which allows leveraging solvers who support the format (namely veriT and CVC4, the latter via a translator [16]) for automatic theorem proving. In Isabelle/HOL in particular this integration has been very successful with the veriT solver, significantly increasing the success rate of the popular Sledgehammer tactic [36]. The format has been refined and extended through the years [6], being now mature and used by multiple systems, with support for core SMT theories, quantifiers, and pre-processing. It allows different levels of granularity, so that solvers can provide coarse-grained proofs (which are easier to produce), or take the effort to produce more detailed, fine-grained proofs (which are often easier to check). It provides a term language close to SMT-LIB [8], facilitating printing from solvers as well as validating the connection between proofs and the corresponding proved problems. An overview of the Alethe proof format is given in Section 2.

A significant drawback of the Alethe format, however, is that it does not have an independent proof checker. This makes it harder for solvers to adopt the format, since to test their proof production they must be directly integrated with the proof assistants with Alethe reconstructions available. Moreover, these reconstruction methods do not check whether proof steps comply to the format's semantics, but rather are used as hints for internal tactics. Finally, the reconstruction techniques struggle with scalability due to well-known performance issues in the proof assistants [12,36].

In this paper we introduce Carcara[3] (Section 3), an independent proof checker for Alethe proofs, implemented in a high-performance programming language, Rust. Carcara is open-source and available under the Apache 2.0 license. Proof checking (Section 3.1) is performed by a collection of modules specific for each rule being checked. The presence of coarse-grained steps in Alethe requires special handling in the checker to account for missing information, which are discussed in detail. Carcara also provides proof elaboration methods (Section 3.2) for particularly impactful coarse-grained steps, so that they can be automatically translated, offline from the solver, into easier-to-check fine-grained steps. We evaluate (Section 4) Carcara's proof checking on a large set of proofs generated by veriT from SMT-LIB problems, analyzing its performance and effectiveness. The same set of proofs is used to evaluate the proof elaboration methods, where we analyze how checking elaborated proofs compares with the

[1] The format was previously known as the "veriT format", but it has recently been renamed to reflect its independence from any individual solver.

[2] cvc5's support for Alethe is still experimental and is under active development. Carcara can actually be instrumental for improving cvc5's support for Alethe.

[3] We follow on the bird theme of the "Alethe" name. Carcará is the Portuguese word for the crested caracara, a resourceful bird of prey native of South America.

originals. Our analysis shows that Carcara has performant proof checking and can identify wrong proofs produced by veriT. It also shows that elaboration can in some cases generate proofs significantly easier to check than the original ones.

1.1 Related work

Carcara is inspired by the highly-successful DRAT-trim [23] proof checker for SAT proofs, which has been instrumental to the extensive usage of proofs in toolchains involving SAT solvers. It has also provided a basis for numerous advances in SAT proofs, with new proof formats and new checking techniques. We see its performant proof checking and elaboration techniques as the key elements to its success, serving both as an independent checker and as a bridge between solvers and performance-critical checkers, such as proof assistants or certified checkers. Providing both these features is the main goal of Carcara.

The checker for the Logical Framework with Side Conditions (LFSC) [37], an extension of Edinburgh's Logical Framework (LF) [22], written in C++, is also a stand-alone, non-certified, highly efficient proof checker. The logical framework, where new rules can be mechanized in a language understood by the checker, provides great flexibility, and LFSC has been successfully used as a proof format for CVC4 [28] and cvc5 [5]. Similarly, Dedukti [25] is an OCaml checker for the $\lambda\Pi$-calculus, another extension of LF, and has been applied to SMT proofs, including to Alethe[4]. However, we are not aware of any mature implementation for this end. Elaboration techniques have not been the focus in these tools. Another difference is that they are based on dependently-typed languages far-removed from SMT-LIB, and generating proofs from SMT solvers for them can be more challenging, as well as relating the resulting proofs to the original problems.

An independent checker has been proposed for SMT proofs [34] from the OpenSMT [26] solver. The checker targets problems with uninterpreted functions and linear arithmetic, but does not support quantifiers nor pre-processing. It leverages DRAT-trim for the propositional reasoning and employs Python components for checking the other parts of the proof. Different components can use different proof formats, and to the best of our knowledge no comprehensive specification of the overall format is available. Some SMT solvers, such as SMT-Interpol [24] and cvc5 [7], have internal checkers for their proofs. Since these are not independent from the solvers, they are incomparable to our approach.

2 The Alethe Proof Format

Alethe was originally designed [10] as a proof-assistant friendly, easy-to-produce proof format for SMT solvers. A clear specification of the rules in a reference document [2] is provided, facilitating reconstruction within proof assistants by avoiding ambiguous syntax or semantics. To facilitate proof production, Alethe uses a term language that directly extends SMT-LIB, thus not requiring solvers

[4] "Verine" library available at https://deducteam.github.io/data/libraries/verine.tar.g

to translate between different term languages when outputting proofs. More importantly, Alethe's proof calculus provides rules with varying levels of granularity, allowing coarse-grained steps and relying on powerful proof checkers for filling in the gaps. This reduces the burden on developers to track all reasoning steps performed by the solver, a notoriously difficult task [7]. The set of rules in the format captures SMT solving (as generally performed by CDCL($\mathcal{T}$)-based SMT solvers [31]) for problems containing a mix of any of quantifiers, uninterpreted functions, and linear arithmetic, as well as multiple pre-processing techniques. As a testament of the format's success, it has been refined and extended throughout the years [6], and has been used as the basis for the integration, with the proof assistants Isabelle/HOL and Coq, of the SMT solvers veriT [6,36], CVC4 [16] and cvc5 [5, Sec. 3].

Here we briefly overview the Alethe proof format. For the full description of its syntax and semantics please see [2]. We assume the reader is familiar with basic notions of many-sorted equational first-order logic [17]. Alethe proofs have the form $\pi : \varphi_1 \wedge \cdots \wedge \varphi_n \to \bot$, i.e., they are refutations, where $\bot$ is derived from assumptions $\varphi_1, \ldots, \varphi_n$ corresponding to the original SMT instance being refuted. Proofs are a series of steps represented as an indexed list of `step` commands. The command `assume` is analogous to `step` but used only for introducing assumptions. The indexed steps induce a directed acyclic graph rooted on the step concluding $\bot$ and with the assumptions $\varphi_1, \ldots, \varphi_n$ as leaves. Steps represent inferences and abstractly have the form

$$c_1, \ldots, c_k \rhd i.\ \psi_1, \ldots, \psi_l\ (\texttt{rule}\ p_1, \ldots, p_n)\ [a_1, \ldots, a_m]$$

where `rule` names the inference rule used in this step. Every step has an identifier i and concludes a clause, represented as a list of literals $\psi_1, \ldots, \psi_l$. The premises are identifiers $p_1, \ldots, p_n$ of previous steps or assumptions, and rule-dependent arguments are terms $a_1, \ldots, a_m$; steps may occur under a *context*, which is defined by bound variables or substitutions $c_1, \ldots, c_k$. Contexts are introduced by the `anchor` command, which opens subproofs. Subproofs simulate the effect of the $\Rightarrow$-introduction rule of Natural Deduction, where local assumptions are put in context and the last step in a subproof represents its conclusion and the closing of its context. Besides arbitrary formulas, Alethe has support for contexts which put in scope bound variables and substitutions, which are useful for representing pre-processing techniques in the presence of binders [6], such as Skolemization, let elimination and alpha-conversion.

The structure of Alethe proofs is motivated by SMT solvers generally operating with a cooperation of a SAT solver and multiple engines to perform theory reasoning, deriving new facts and applying simplifications. The overall proof may be seen as a ground first-order resolution proof with theory lemmas justified by closed subproofs. Thus the emphasis on steps concluding clauses as term lists, which avoids ambiguity as to what clause a disjunction represents. An example is that whether a resolution step concluding the term $A \vee B$ corresponds to the clause $[A, B]$ or $[A \vee B]$ depends on the premises. The use of identifiers for steps allows representing proofs as directed acyclic graphs rather than trees. Similarly,

```
(set-logic LIA)
(assert (forall ((x Int)) (> x 0)))
(assert (not (forall ((y Int)) (> y 0))))
(check-sat)
```

```
(assume h1 (forall ((x Int)) (> x 0)))
(assume h2 (not (forall ((y Int)) (> y 0))))
(anchor :step t3 :args ((y Int) (:= x y)))
(step t3.t1 (cl (= x y)) :rule refl)
(step t3.t2 (cl (= (> x 0) (> y 0))) :rule cong :premises (t3.t1))
(step t3 (cl (= (forall ((x Int)) (> x 0)) (forall ((y Int)) (> y 0))))
  :rule bind)
(step t4 (cl (not (forall ((x Int)) (> x 0))) (forall ((y Int)) (> y 0)))
  :rule equiv1 :premises (t3))
(step t5 (cl) :rule resolution :premises (t4 h1 h2))
```

Fig. 1: A simple SMT-LIB problem and an Alethe proof of its unsatisfiability.

term sharing can be achieved via the SMT-LIB `:named` attribute or `define-fun` commands [8, Sec1 4.1.6], which both allow naming subterms. These measures are essential for compact representation of proofs, which can be prohibitively large otherwise. Explicitly providing the conclusion of proof steps aims to both facilitate proof checking (as it allows steps to be verified locally) and proof production, so coarse-grained rules that do not uniquely define their conclusions from premises and arguments can be effectively checked.

Example 1. Figure 1 shows an SMT-LIB problem and an Alethe proof of its unsatisfiability. Note that in Alethe's concrete syntax clauses are represented via the `cl` operator (the only exception are conclusions of `assume` commands, which are considered unit clauses) and the context is not explicitly put in the steps, but rather assumed for all steps under (potentially nested) anchors introducing its elements. For this proof to be valid, three conditions need to be met: each `assume` command must correspond to an `assert` command in the original problem, every `step` command must be valid according to the semantics of its rule, and the proof must end with a step that concludes the empty clause `(cl)`. The proof satisfies the first condition, as the terms in the `assume` commands are precisely the asserted terms in the SMT problem. The third condition holds as `t5`, the last step, concludes the empty clause. For the second condition, step `t4` is a direct consequence of the equivalence in its premise, `t3`, so it remains to check step `t3`, which is derived from a subproof. The anchor for `t3` introduces a bound variable y and a substitution $\{x \mapsto y\}$. The steps in the subproof contain terms with this new variable and operate under this substitution. The rule `refl` models reflexivity modulo the cumulative, capture-avoiding substitution in the (potentially nested) context, and thus `t3.t1` holds since $x = y\{x \mapsto y\}$. Step `t3.t2` is regular congruence with the operator ">" and does not depend on the context. Finally, step `t3` holds because its subproof shows the equivalence of the

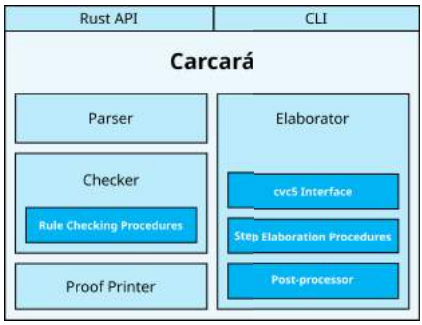


Fig. 2: Overview of the architecture of Carcara.

bodies of the quantifiers under the renaming, introduced in the context, into a fresh variable relative to the left-hand side quantifier. Since all steps follow the expected semantics, all conditions are met and the proof is valid.

In the next section we show how Carcara checks the above conditions, highlighting some challenging rules and showing how some coarse-grained steps are elaborated into proofs potentially simpler to check.

3 Architecture and core components

Carcara is developed in the Rust programming language, and is publicly available[5] under the Apache 2.0 license. Its architecture is shown in Figure 2. It provides both a command line interface and bindings for a Rust API. The main component is the proof checking one, with 6.5k LOC, which is a collection of procedures for each rule to be checked (Section 3.1). The elaborator has 1k LOC and has an interface to the cvc5 solver, as well as a collection of elaboration methods and a post-processing module to knit together the elaborated proof (Section 3.2). The other components together have 6k LOC, including a handwritten 2k LOC SMT-LIB and Alethe parser, and an Alethe printer.

The inputs of Carcara are an SMT-LIB problem φ and an Alethe proof $\pi : \varphi \to \bot$. In proof-checking mode it checks each step in π with the respective procedure for its rule and prints either `valid`, when all steps are successfully checked and the proof concludes the empty clause `(cl)`, `holey` when π is valid but contains steps that are not checked ("holes"), and `invalid` otherwise, together with an error message indicating the first step where checking failed and why. In proof-elaboration mode it converts π into $\pi' : \varphi \to \bot$, where some steps may be replaced by a series of steps elaborating them, and prints π'.

[5] https://github.com/ufmg-smite/carcara

3.1 Checking Alethe proofs

First the original SMT-LIB problem and its Alethe proof are parsed. The problem provides the declaration of sorts and symbols that may be used in the proof, as well as the original assertions, which must match the assumptions in the proof. Symbol definitions in the proof for term sharing are expanded during parsing. Terms are internally represented as directed acyclic graphs, using *hash consing* for maximal sharing and constant-time syntactically-equality tests. The proof is represented internally as an array of command objects, each corresponding either to an Alethe `assume` or `step` command, or a subproof, which is represented as a step with an (arbitrarily) nested array of command objects. Step identifiers are converted into indices for the arrays, so that access is constant-time.

Each command is checked individually by the rule checker corresponding to the rule in that command. That component takes as input the conclusion, the conclusions of its premises, and the arguments of the command, as well as the context it is in. As the Alethe format currently has 90 possible rules, Carcara has 90 rule checkers. We highlight below some of the rule checkers as well as some challenges for checking Alethe proofs and how we addressed them.

Term equality tests. Terms introduced by Alethe rules may have equality subterms implicitly reordered, but the rules are still valid if the conclusion changes only in this way. This flexibility is motivated by solvers often internally representing equalities ignoring order, which may lead to equalities being implicitly reordered when appearing in facts derived by these components. The congruence closure procedure [29] commonly used in SMT is an example of such a component. Since equality symmetry justifies these reorderings, but keeping track of all the changes can be challenging, the format allows them to be implicit.

As a consequence, syntactic equality cannot be the only test for whether two terms are the same. For example, the terms `(and p (= a b))` and `(and p (= b a))` may be required to be equal. Thus Carcara tests equality in two phases: first if they are syntactically equal, in which case they can be compared in constant time; otherwise they are simultaneously traversed and equality subterms in the same position are compared modulo equality reordering, failing as soon as subterms differ. We refer to this as a *polyequal* test. As we will see in Section 4.1, these tests can be a substantial portion of overall checking time in some cases.

Checking initial assumptions. The initial `assume` commands in an Alethe proof must correspond to assertions in the original problem, so their checker searches through the assertions to find a match. In general, this can be done efficiently: assertions are stored in a hash set during parsing, and these `assume` commands are valid if their conclusions occur in the set. However, `assume` commands are also impacted by implicit equality reordering, thus requiring polyequal tests. When an assumption does not occur in the assertions hash set, the checker attempts to match it to each assertion in turn, performing a polyequal test. As a consequence, when the original problem is large and the assertions similar and deep, checking `assume` steps may dominate overall checking time, as our experiments show (Section 4.1).

Checking contextual steps. Steps within subproofs may depend on their context to be valid, so before checking these steps, a context object is built based on the `anchor` opening the subproof. As shown in Section 2, context elements on which rules may depend are bound variables and substitutions. The former make new symbols available to build terms, while the latter allows steps to be valid modulo applying these substitutions.

Substitutions in Alethe are capture-avoiding, renaming bound variables during application, which facilitates producing proofs with binders [6]. However, it has the side effect of also preventing constant-time equality tests, since we must rather check α-equivalence, i.e., a term with bound variables may be required to be equal[6] to the result of applying a substitution that may have renamed some of these variables. To avoid spurious renaming when applying substitutions, the checker only renames bound variables which occur as free variables in the substitution range. Since computing free variables is itself costly, it is done lazily, only when the substitution is to be applied under a binder, and the result is cached.

Note that, as subproofs can be nested, the substitution in context for a step is the composition of a stack of substitutions $\sigma_1, \ldots, \sigma_n$. To avoid sequential application of substitutions, Alethe requires the substitution σ in context to be a cumulative substitution in which every term t in the range of the substitution σ_{i+1} is replaced by $t\sigma_i$. Thus σ can be applied simultaneously and correspond to a sequential application of $\sigma_1, \ldots, \sigma_n$. As a result of these requirements, handling and applying substitutions can be expensive in Alethe, as shown in Section 4.1.

Finally, the rules enclosing subproofs must be checked to whether their conclusions are valid from the introduced context and resulting subproof. For example, the `bind` rule in Example 1 requires that the bound variable in the quantifier at the right-hand side of the equality matches the range of the substitution put in context for its subproof. The `subproof` rule, which introduces local assumptions $a_1, \ldots, a_n$, and concludes a formula $\neg a'_1 \vee \cdots \neg a'_n \vee \varphi$, requires that the enclosed subproof derives φ and that each a_i match a'_i.

We now highlight coarse-grained rules whose checking is more intricate and expensive.

Resolution. The rule `resolution` in Alethe captures hyper-resolution on ground first-order clauses, i.e.,

$$\frac{C_1 \quad \cdots \quad C_n}{C}\ \texttt{resolution}, p_1, p_2, \ldots, p_{n-1}$$

where $C_1, \ldots, C_n$ are premises; p_i the pivot for the binary resolution between C_i and C_{i+1}, occurring as is in C_i and as $\neg p_i$ in C_{i+1}; and C the conclusion. While it is simple to check such steps, Alethe allows `resolution` steps to not provide the pivots, for the sake of facilitating proof-production in solvers. Checking such steps requires searching for the pivots and in which binary resolution they are to

[6] Since Alethe has bound-variable renaming rules, the checker requires names to be handled properly, rather than normalizing all binders internally via De Brujin indices.

be used, but Carcara applies an incomplete heuristic where pivots are inferred between the difference of literals in the premises and in the conclusion (i.e., literals not in the conclusion must have been pivots eventually eliminated). If that fails, we apply a reverse unit propagation (RUP) test [21], i.e., the step is valid if we can derive a conflict via Boolean Constraint Propagation from the premises and the negated conclusion. Note that Carcara also allows the pivots to be provided as arguments, in which case checking is simple, as expected.

AC simplification. Normalization modulo associativity and commutativity for conjunction and disjunction can be represented in Alethe via the `ac_simp` rule, which establishes the equality between a term t and a term t' that is t but with nested occurrences of these connectives flattened and duplicate arguments removed, until a fix-point. While this simplification is performance-critical [6, Sec. 4.6], checking the corresponding rule requires traversing t and performing the normalization, which is proportional to t's depth.

Arithmetic reasoning. Apart from simplification rules, arithmetic reasoning in Alethe is mainly captured by two rules: `la_generic` and `lia_generic`. Both rules conclude a clause of negated linear inequalities, which is valid due to the Farkas' lemma [18] guaranteeing that there exists a linear combination of these inequalities equivalent to $\bot$. The `la_generic` rule takes as arguments the coefficients of this linear combination, with which the rule can be checked by applying simple (but costly) operations on the coefficients to reduce the linear combination to $\bot$ (see [2, Sec 5.4, Rule 9] for the algorithm). The checker uses GMP [1] for efficiently performing the required computations with the coefficients.

While `la_generic` can be checked effectively, `lia_generic` cannot. It provides only the negated inequalities, which would require searching for the coefficients to perform the checking, essentially requiring the arithmetic solving to be repeated in the checker. As a consequence this rule is considered a hole and Carcara ignores it during proof checking, issuing a warning.

3.2 Elaborating Alethe proofs

In order to mitigate bottlenecks in checking some Alethe steps, Carcara can also elaborate Alethe proofs into easier-to-check ones by filling in missing details from the original proofs. This is done by replacing coarse-grained steps with fine-grained proofs of their conclusions, producing a new overall proof equivalent to the original, but with some coarse-grained steps broken down into fine-grained ones. Formally, a proof as the one below on the left, with a coarse step concluding ψ from premises $\psi_1, \ldots, \psi_n$, is elaborated into the proof on the right where the coarse step is replaced by a proof π, with *fine-grained* steps, rooted on ψ and with $\psi_1, \ldots, \psi_n$ as leaves:

$$\dfrac{\dfrac{\psi_1 \;\cdots\; \psi_n}{\psi}\;\textsc{coarseStep} \qquad \cdots}{\Theta}\;\textsc{rule} \quad \Rightarrow_{\text{elab}} \quad \dfrac{\dfrac{\dfrac{\psi_1 \;\cdots\; \psi_n}{\pi}}{\psi} \qquad \cdots}{\Theta}\;\textsc{rule}$$

```
(step t2.t1 (cl (not (= a b)) (not (= b c)) (not (= c d)) (= a d))
  :rule eq_transitive)
(step t2.t2 (cl (not (= b a)) (= a b)) :rule eq_symmetric)
(step t2.t3 (cl (not (= c b)) (= b c)) :rule eq_symmetric)
(step t2.t4 (cl (not (= c d)) (= a d) (not (= b a)) (not (= c b)))
  :rule resolution :premises (t2.t1 t2.t2 t2.t3))
(step t2 (cl (not (= b a)) (not (= c d)) (not (= c b)) (= a d))
  :rule reordering :premises (t2.t4))
```

Fig. 3: Elaboration of an `eq_transitive` step. Note the new `eq_transitive` step is easy to check, and the new `t2` step has the same conclusion as the original.

Note the expansion only affects the proof locally, since any step using the conclusion of the coarse step as a premise may use the conclusion of π interchangeably.

There are many Alethe rules whose checking would be simpler if elaborated, but we have focused initially on what we believe can be more impactful: removing implicit equality reordering, and thus polyequal tests, which affects virtually every Alethe rule; and providing checkable justifications for `lia_generic` steps, to remove holes from proofs. Before detailing these methods, we illustrate the elaboration process with an example.

Elaborating transitivity steps. The `eq_transitive` rule concludes a valid clause composed of negated equalities followed by a single positive equality, such that the negated equalities form a transitive chain resulting in the final equality. However, the specification does not impose an order on the negated equalities (which can, remember, also be implicitly reordered). So the following step must also be valid, with a "shuffled" chain:

```
(step t2 (cl (not (= b a)) (not (= c d)) (not (= c b)) (= a d))
  :rule eq_transitive)
```

This permissive specification again facilitates proof production (particularly from congruence closure procedures), but requires the `eq_transitive` checker, for every link in the chain, to potentially traverse the whole clause searching for the next one, performing polyequal tests throughout. The goal of elaborating `eq_transitive` steps is that steps like `t2` are justified in a fine-grained manner. If we changed the conclusion of the step, this would impact the rest of the proof, if `t2` is used anywhere as a premise. We instead introduce a fine-grained proof for `t2`'s conclusion, as shown in Figure 3: an easy-to-check `eq_transitive` step (`t2.t1`), `eq_symmetric` steps to flip the equalities (`t2.t2`, `t2.t3`), `resolution` (`t2.t4`) and `reordering` (`t2.t5`) steps to derive the original conclusion.

Elaborating implicit equality reordering. Similarly to above, steps concluding a term t, with some subterm equality implicitly reordered, have their conclusion replaced by t' where that subterm is not reordered and a fine-grained proof of the conversion of t' into t is added. Figure 4 illustrates this process for an `assume`

```
(set-logic QF_UF)
(declare-const a Bool)
(declare-const b Bool)
(declare-const p Bool)
(assert (not (or p (= a b))))
(assert (or p (= b a)))
(check-sat)
```

Fig. 4a: An example SMT problem instance.

```
(assume h1 (not (or p (= a b))))
(assume h2 (or p (= a b)))
(step t3 (cl) :rule resolution
              :premises (h1 h2))
```

Fig. 4b: An Alethe proof for the SMT problem in Figure 4a. Notice that this proof makes use of implicit reordering of equalities in `h2`.

```
(assume h1 (not (or p (= a b))))
(assume h2 (or p (= b a)))
(step h2.t1 (cl (= (= b a) (= a b))) :rule equiv_simplify)
(step h2.t2 (cl (= (or p (= b a)) (or p (= a b))))
    :rule cong :premises (h2.t1))
(step h2.t3 (cl (not (or p (= b a))) (or p (= a b)))
    :rule equiv1 :premises (h2.t2))
(step h2.t4 (cl (or p (= a b))) :rule resolution :premises (h2 h2.t3))
(step t3 (cl) :rule resolution :premises (h1 h2.t4))
```

Fig. 4c: The elaborated proof without implicit equality reordering.

Fig. 4: An example of the elaboration to remove implicit equality reordering.

command, where note that step `h2.t1` is the rewriting justifying the equality reordering of the subterm and the following steps rebuild the original conclusion.

In the original proof, the `assume` command `h2` introduces the term `(or p (= a b))`, which is the original assertion `(or p (= b a))` with the equality `(= b a)` implicitly reordered. In the elaborated proof (Figure 4c), the conclusion of `h2` is replaced by one without implicit equality reordering, but step `t3` expects the original conclusion. The steps `h2.t1` to `h2.t4` convert the new `h2` conclusion into the original one, relying on standard equality reasoning and on resolution to connect the introduced steps. Notice that the `t3` step, which originally refered to `h2` as a premise, now refers to `h2.t4`.

When applied to every concluding terms with implicit equality reordering, the result of this elaboration method is a proof where equality tests are only syntactic, erasing the overhead of checking assumptions and polyequal tests.

Elaborating `lia_g n ric` steps. As discussed in Section 3.1, Carcara considers `lia_generic` steps holes in the proof, as their checking is as hard as solving. Since our goal is to keep Carcara as simple as possible, we rely on an external tool to elaborate the step by solving a problem corresponding to it in a proof-producing manner, then import the proof, checking it and guaranteeing that it is sound to replace the original step. Any tool producing detailed Alethe proofs for linear-integer arithmetic reasoning can be used to this end, but currently only cvc5 can do so [7]. We note that cvc5 currently has the limitation that its Alethe

proofs may contain rewrite steps not yet modeled in the Alethe simplification rules [2, Sec 5.11], and are thus not supported by Carcara. They are considered holes, but since these are generally simple simplification rules, are much less harmful than `lia_generic` ones.

In detail, the elaboration method, when encountering a `lia_generic` step S concluding the negated inequalities $\neg l_1 \vee \cdots \vee \neg l_n$, generates an SMT-LIB problem asserting $l_1 \wedge \cdots \wedge l_n$ and invokes cvc5 on it, expecting an Alethe proof $\pi : (l_1 \wedge \cdots \wedge l_n) \to \bot$. Carcara will check each step in π and, if they are not invalid, will replace step S in the original proof by a proof of the form:

```
(anchor :step S.t_m+1)
(assume S.h_1 l1)
...
(assume S.h_n ln)
...
(step S.t_m (cl false) :rule ...)
(step S.t_m+1 (cl (not l1) ... (not ln) false) :rule subproof)
(step S.t_m+2 (cl (not false)) :rule false)
(step S (cl (not l1) ... (not ln))
  :rule resolution :premises (S.t_m+1 S.t_m+2))
```

where steps `S.h_1` until `S.t_m` are imported from the cvc5 proof. As a result the `lia_generic` step `S` in the original proof will have been replaced by a detailed justification whose correctness can be independently established by Carcara.

4 Evaluation

We evaluate Carcara for proof-checking performance and the impact of elaboration methods. We use the veriT solver [13], version 2021.06-40-rmx, to generate Alethe proofs from all problems in the SMT-LIB benchmark library[7] whose logic it supports, with a 120 seconds timeout. We did not consider cvc5 as its support for Alethe is not yet as mature or complete. The veriT solver produced 39,229 proofs. They total 92gb, but vary greatly in size. The biggest proof has 4.5gb, fourteen have at least 1gb and over a hundred have more than 100mb, while almost 90% are under 1mb. All the experiments were run on a server equipped with AWS Graviton2 2.5 GHz ARM CPUs, with 4 GB of memory for each job.

4.1 Proof checking

We ran Carcara on each proof until checking succeeded or failed. Only 378 had checking failures, which were due to incorrect[8] steps for quantifier simplifications (Skolemization and elimination of one-point quantifiers) and AC normalization. The issues have been communicated to the solver developers. For the successful proofs, a summary is given in Table 1, for each SMT-LIB logic, with the cumulative solving time by veriT and checking time by Carcara.

[7] https://smtlib.cs.uiowa.edu/benchmarks.shtml

[8] In a superficial analysis the steps seemed sound, but the proofs were incorrect.

Logic	Problems	Solving time (s)	Checking time (s)	Ratio
AUFLIA	2135	1094.67	12.51	87.53
AUFLIRA	19200	248.95	144.03	1.73
UF	2885	2858.14	30.95	92.35
UFIDL	55	0.54	0.66	0.82
UFLIA	7221	3547.78	136.21	26.05
UFLRA	10	0.02	0.01	3.05
QF_ALIA	16	0.79	1.39	0.57
QF_AUFLIA	256	0.34	0.11	3.04
QF_IDL	609	3316.08	2240.10	1.48
QF_LIA	1018	5975.36	742.73	8.05
QF_LRA	537	3629.39	258.60	14.03
QF_RDL	81	620.46	123.14	5.04
QF_UF	4180	3857.34	1881.55	2.05
QF_UFIDL	66	396.74	87.58	4.53
QF_UFLIA	167	1194.51	4.70	254.41
QF_UFLRA	415	141.82	65.14	2.18
Total:	38851	26882.93	5729.39	4.69

Table 1: Total solving and proof-checking time per logic for veriT and Carcara.

As expected, the comparison is heavily logic-dependent. In quantified logics (top of the table), checking is generally significantly cheaper than solving. An outlier is AUFLIRA, which is explained by the problems to which veriT could produce proofs being all both simple to solve and check. In logics such as QF_UF and QF_IDL, which can have very large proofs, overall checking time is comparable to solving time, if still noticeably smaller in total.

When comparing per-problem, for the large majority of proofs (81.61%) the checking time was smaller than the solving time. Furthermore, for 3.96% of the proofs, checking was more than 10 times faster than solving the problem, and for 0.96%, that ratio was of 100 times. There were only 24 instances where the checking time was more than 10 times bigger than the solving time, and, in all of them, the checking time was less than 0.6 seconds.

We also evaluate the per-rule frequency, as shown in Figure 5b, and checking time, with Figure 6a showing the cumulative checking times and Figure 5a a box plot considering individual rule checks. The lower whisker represents the 5th percentile, the lower bound of the box represents the first quartile, the line inside the box represents the median, the upper bound of the box represents the third quartile, and the upper whisker represents the 95th percentile[9]. Rules that are rare and have negligible checking time are omitted. The data is gathered from proof checking in all proofs, even those that failed.

The `assume` commands account for a large proportion of the total time. This is justified by their checking, due to implicit equality reordering, being potentially proportional to both the quantity and the depth of assertions in the original problems. The box plot shows that the worse cases lead to the most expensive rule checks among all rules.

[9] The plots follow the same criteria of the evaluation in [36].

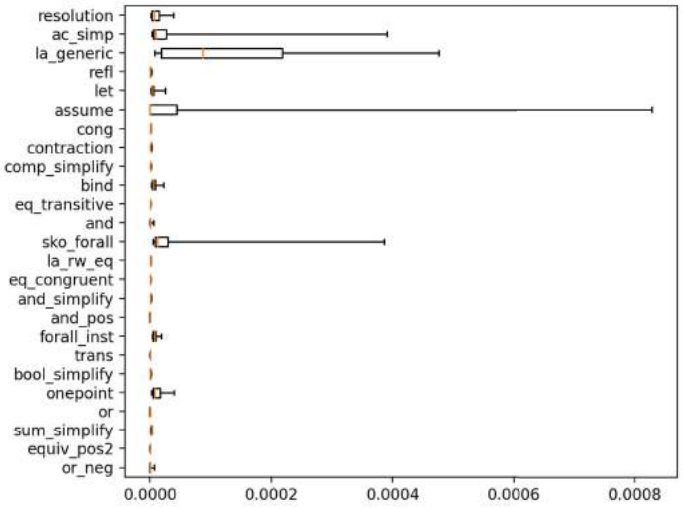


Fig. 5a: Box plot for checking time per rule.

Rule	%
cong	31%
resolution	27%
refl	17%
comp_simplify	5%
eq_transitive	4%
la_rw_eq	2%
ac_simp	1%
and_pos	1%
and	1%
bind	1%
trans	< 1%
or	< 1%
equiv_pos2	< 1%
eq_congruent	< 1%
la_generic	< 1%
...	< 1%

Fig. 5b: Perc. of total steps per rule (only most frequent shown).

Rules with highest overall time are `resolution`, `ac_simp` and `la_generic`. For `resolution` this is explained mainly by its high frequency (this is similarly the case for `cong`), as well as by some more expensive checks (veriT does not provide pivots), as shown in the box plot. As for `ac_simp` and `la_generic`, while they are much less frequent, their checking is expensive (Section 3.1).

Other expensive rules to note are those related to contexts involving substitutions[10], specially `let`, for let elimination, and `refl`. It is common for `let` subproofs to be deeply nested, leading to large cumulative substitutions needing to be computed. As for `refl`, besides being one of the most frequent rules, about a third of its total time is spent on polyequal tests, and most of the rest is related to handling and applying substitutions, as well as checking alpha-equivalence.

4.2 Proof elaboration

We ran Carcara, on each successfully checked proof, in proof-elaboration mode with the elaboration of transitivity steps and, more importantly, the removal of implicit equality reordering. On average, excluding parsing, elaboration takes 40% of the time required for checking. We focus on the impact on proof checking of the result of elaboration.

In Figure 7 we have the comparison, per proof, of the proof-checking time on the original proof and on the elaborated one (excluding parsing time). There is not a clear winner, but note that for harder proofs (those originally requiring at least 1s), checking the elaborated proof is often significantly faster. A per-rule analysis is shown in Figure 6b, with the proportion of the checking time spent

[10] The ones shown in the plots are `let`, `bind`, `sko_forall`, and `onepoint`.

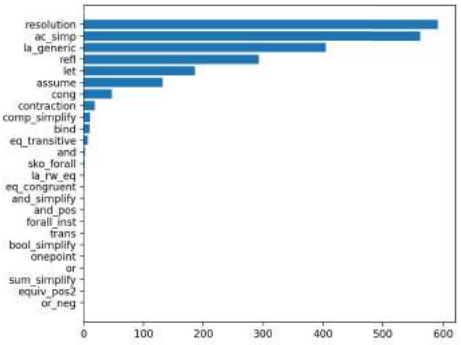


Fig. 6a: Total checking time per rule.

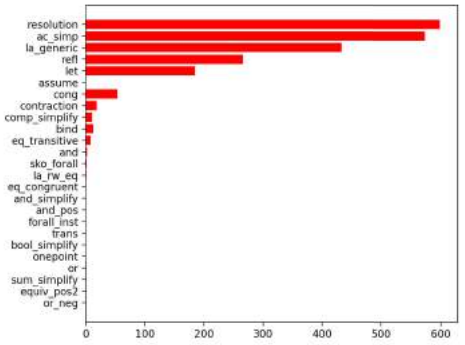


Fig. 6b: Times after elaboration.

in each rule, for the elaborated proofs. Comparing to Figure 6a, the checking time for `assume` steps becomes negligible in the elaborated proofs, as checking them now amounts to checking occurrence in a hash set. The overall time for `refl` also decreases, but only by 10%. This can be explained by the `refl` steps added during elaboration. While checking each `refl` is now potentially cheaper, this is offset by their increased number. Note that these additions also impact other rules, specially `cong`, whose cumulative time increased by 13%. Overall, proof elaboration resulted in a net improvement in checking time of 6%. Parsing time, however, increased, which made the overall runtime for proof-checking the original proofs virtually the same as for the elaborated proofs.

The results indicate that elaborating implicit equality reordering is not always worth it, specially for high-performant tools. However, it successfully yields proofs not requiring polyequal tests, which may help performance in other scenarios. For example, the reconstruction of Alethe proofs in Isabelle/HOL requires equality tests to be done by applying a normalizer to both terms and then testing them for syntactic equality. This leads to performance issues for reconstructing some rules [36], which this elaboration method would avoid.

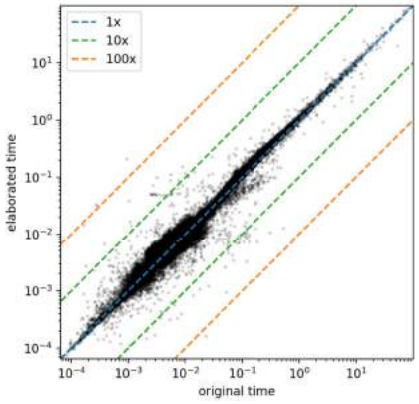


Fig. 7: Before vs after elaboration.

Elaborating `lia_generic` steps. In our benchmark set, 276 proofs contain a total of 127k `lia_generic` steps. As a proof of concept we instrumented CARCARA to apply the elaboration method described in Section 3.2 via a connection with cvc5[11]. Due to the still experimental Alethe proof production in cvc5, we only considered SMT problems derived

[11] cvc5-1.0.2, modified for better Alethe support, provided by the cvc5 team.

from `lia_generic` steps in proofs for the QF_UFLIA and QF_LIA logics. This excluded only 15 proofs, each containing exactly one `lia_generic` step. We ran Carcara on proof-elaboration mode with a 30 minute timeout for each proof. For each `lia_generic` step, cvc5 was invoked with a 30s timeout and the resulting Alethe proof, if any, replaced the original `lia_generic` step, as described in Section 3.2.

Of the 261 proofs, Carcara timed out on only 13 of them. Of the remaining 248 proofs, 82 still contained `lia_generic` steps after elaboration, either because cvc5 timed out when solving the generated problem, or because the cvc5 proofs contained `lia_generic` steps of their own. Note however that they are still improvements over the original `lia_generic` steps, since generally less inequalities are involved and the steps are potentially simpler to solve, were the process to be repeated. Similarly, although all elaborated proofs contained holes from cvc5 rewriting steps, these are much simpler than the original `lia_generic` ones.

As with the elaboration of implicit equality reordering, this elaboration method would be particularly impactful in scenarios such as Alethe reconstruction in Isabelle/HOL. Steps such as `lia_generic` are reconstructed via limited internal automation for arithmetic reasoning, which is known to fail [36, Sec. 4.3].

5 Conclusion and future work

Our evaluation shows that Carcara has good performance and can identify shortcomings in the proof-production of established SMT solvers. Carcara can also elaborate proofs into demonstrably easier-to-check ones, which can have a significant impact, for example, if it is used as a bridge between solvers and proof assistants. Extending Carcara to convert Alethe proofs into other formats would also allow the elaboration techniques to benefit other toolchains.

As future work, we will add support for parallel proof checking, since steps in the same context can be checked completely independently. We will also add new elaboration methods for `resolution` and `ac_simp`, which occasionally are bottlenecks, and will provide elaboration for rewrite rules, which can change significantly between different solvers, complicating proof-production if solvers have to phrase their rewrites with a fixed set of rules. An automatic conversion into a defined set of rewrite rules, as described in [32], would address this issue.

Finally, we expect Carcara to facilitate improving how we use Alethe proofs. For example, our large-scale evaluation shows the significant time spent on contextual substitutions, which is mainly due to the Alethe requirement of only applying substitutions simultaneously. Extending the proof format to allow other substitution application strategies may be beneficial for different scenarios, as proof production in some solvers has indicated [7, Sec 5.1]. In general, extensions to the format (for example, to other logical theories) can be done in a more informed way with the help of an independent checker.

Acknowledgments. We thank the reviewers for their helpful suggestions to improve this paper as well as Carcara. We thank Hans-Jörg Schurr for his extensive work in detailing the semantics of Alethe, which greatly facilitated developing Carcara.

Data Availability Statement. The datasets generated and analyzed during the current study are available in the Zenodo repository: https://zenodo.org/record/7574451 [3].

References

1. GNU Multiple Precision Arithmetic Library. http://gmplib.org/, Oct 2022.
2. The Alethe Proof Format: A Speculative Specification and Reference. https://verit.loria.fr/documentation/alethe-spec.pdf, Oct 2022.
3. Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. Carcara artifact, 2023. zenodo, https://doi.org/10.5281/zenodo.7574451.
4. Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.
5. Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS), Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.
6. Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, and Pascal Fontaine. Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning*, 64(3):485–510, 2020.
7. Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark W. Barrett. Flexible proof production in an industrial-strength SMT solver. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2022.
8. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
9. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.
10. Frédéric Besson, Pascal Fontaine, and Laurent Théry. A flexible proof format for SMT: a proposal. In *Workshop on Proof eXchange for Theorem Proving (PxTP)*, 2011.
11. Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.
12. Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. Reconstruction of z3's bit-vector proofs in HOL4 and isabelle/hol. In Jean-Pierre Jouannaud

and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2011.
13. Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In Renate A. Schmidt, editor, *Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.
14. Lilian Burdy and David Déharbe. Teaching an old dog new tricks - the drudges of the interactive prover in atelier B. In Michael J. Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings*, volume 10817 of *Lecture Notes in Computer Science*, pages 415–419. Springer, 2018.
15. Leonardo Mendonça de Moura and Nikolaj Bjørner. Proofs and refutations, and Z3. In Piotr Rudnicki, Geoff Sutcliffe, Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) Workshops*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
16. Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. Smtcoq: A plug-in for integrating SMT solvers into coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.
17. Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 2 edition, 2001.
18. G. Farkas. A Fourier-féle mechanikai elv alkamazásai. *Mathematikaiés Természettudományi Értesíto*, 12:457–472, 1894. reference from Schrijver's Combinatorial Optimization textbook (Hungarian).
19. Mathias Fleury. Optimizing a verified SAT solver. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2019.
20. Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using imperative HOL. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 158–171. ACM, 2018.
21. Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*, 2008.
22. Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.
23. Marijn J. H. Heule. The DRAT format and drat-trim checker. *CoRR*, abs/1610.06229, 2016.
24. Jochen Hoenicke and Tanja Schindler. A simple proof format for SMT. In David Déharbe and Antti E. J. Hyvärinen, editors, *International Workshop on Satisfiability Modulo Theories (SMT)*, volume 3185 of *CEUR Workshop Proceedings*, pages 54–70. CEUR-WS.org, 2022.
25. Gabriel Hondet and Frédéric Blanqui. The new rewriting engine of dedukti (system description). In Zena M. Ariola, editor, *International Conference on Formal*

Structures for Computation and Deduction (FSCD), volume 167 of *LIPIcs*, pages 35:1–35:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
26. Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. Opensmt2: An SMT solver for multi-core and cloud computing. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *Lecture Notes in Computer Science*, pages 547–553. Springer, 2016.
27. Shuanglong Kan, Anthony Widjaja Lin, Philipp Rümmer, and Micha Schrader. Certistr: a certified string solver. In Andrei Popescu and Steve Zdancewic, editors, *Certified Programs and Proofs (CPP)*, pages 210–224. ACM, 2022.
28. Guy Katz, Clark W. Barrett, Cesare Tinelli, Andrew Reynolds, and Liana Hadarean. Lazy proofs for dpll(t)-based SMT solvers. In Ruzica Piskac and Muralidhar Talupur, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, pages 93–100. IEEE, 2016.
29. Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, 1980.
30. Aina Niemetz, Mathias Preiner, and Clark W. Barrett. Murxla: A modular and highly extensible API fuzzer for SMT solvers. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification (CAV), Part II*, volume 13372 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2022.
31. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006.
32. Andres Nötzli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. Reconstructing fine-grained proofs of complex rewrites using a domain-specific language. In Alberto Griggio and Neha Rungta, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, 2022. To appear.
33. Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. versat: A verified modern SAT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7148 of *Lecture Notes in Computer Science*, pages 363–378. Springer, 2012.
34. Rodrigo Otoni, Martin Blicha, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. Theory-specific proof steps witnessing correctness of SMT executions. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 541–546. IEEE, 2021.
35. Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In Chantal Keller and Mathias Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021.
36. Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In André Platzer and Geoff Sutcliffe, editors, *Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 450–467. Springer, 2021.
37. Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. SMT proof checking using a logical framework. *Formal Methods in System Design*, 42(1):91–118, 2013.
38. Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 718–730. ACM, 2020.

Constraint Solving/Blockchain

The Packing Chromatic Number of the Infinite Square Grid is 15*

Bernardo Subercaseaux and Marijn J. H. Heule

Carnegie Mellon University, Pittsburgh, PA 15203, USA
{bsuberca,mheule}@cs.cmu.edu

Abstract. A packing k-coloring is a natural variation on the standard notion of graph k-coloring, where vertices are assigned numbers from $\{1, \ldots, k\}$, and any two vertices assigned a common color $c \in \{1, \ldots, k\}$ need to be at a distance greater than c (as opposed to 1, in standard graph colorings). Despite a sequence of incremental work, determining the packing chromatic number of the infinite square grid has remained an open problem since its introduction in 2002. We culminate the search by proving this number to be 15. We achieve this result by improving the best-known method for this problem by roughly two orders of magnitude. The most important technique to boost performance is a novel, surprisingly effective propositional encoding for packing colorings. Additionally, we developed an alternative symmetry breaking method. Since both new techniques are more complex than existing techniques for this problem, a verified approach is required to trust them. We include both techniques in a proof of unsatisfiability, reducing the trusted core to the correctness of the direct encoding.

Keywords: Packing coloring · SAT · Verification.

1 Introduction

Automated reasoning techniques have been successfully applied to a variety of coloring problems ranging from the classical computer-assisted proof of the *Four Color Theorem* [1], to progress on the *Hadwiger-Nelson problem* [21], or improving the bounds on Ramsey-like numbers [19]. This article contributes a new success story to the area: we show the *packing* chromatic number of the infinite square grid to be 15, thus solving via automated reasoning techniques a combinatorial problem that had remained elusive for over 20 years.

The notion of *packing coloring* was introduced in the seminal work of Goddard et al. [10], and since then more than 70 articles have studied it [3], establishing it as an active area of research. Let us consider the following definition.

Definition 1. *A packing k-coloring of a simple undirected graph $G = (V, E)$ is a function f from V to $\{1, \ldots, k\}$ such that for any two distinct vertices $u, v \in V$, and any color $c \in \{1, \ldots, k\}$, it holds that $f(u) = f(v) = c$ implies $d(u, v) > c$.*

* Both authors are supported by the U.S. National Science Foundation under grant CCF-2015445.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 389–406, 2023.
https://doi.org/10.1007/978-3-031-30823-9_20

Note that by changing the last condition to $d(u, v) > 1$ we recover the standard notion of coloring, thus making packing colorings a natural variation of them. Intuitively, in a packing coloring, *larger* colors forbid being reused in a larger region of the graph around them. Indeed, packing colorings were originally presented under the name of *broadcast coloring*, motivated by the problem of assigning broadcast frequencies to radio stations in a non-conflicting way [10], where two radio stations that are assigned the same frequency need to be at distance greater than some function of the power of their broadcast signals. Therefore, a large color represents a powerful broadcast signal at a given frequency, that cannot be reused anywhere else within a large radius around it, to avoid interference. Minimizing the number of colors assigned can thus be interpreted as minimizing the pollution of the radio spectrum. The literature has preferred the name *packing coloring* ever since [3].

Analogously to the case of standard colorings, we can naturally define the notion of *packing chromatic number*, and study its computation.

Definition 2. *Given a graph $G = (V, E)$, define its packing chromatic number $\chi_\rho(G)$ as the minimum value k such that G admits a packing k-coloring.*

Example 1. Consider the infinite graph with vertex set $\mathbb{Z}$ and with edges between consecutive integers, which we denote as $\mathbb{Z}^1$. A packing 3-coloring is illustrated in Figure 1. On the other hand, by examination one can observe that it is impossible to obtain a packing 2-coloring for $\mathbb{Z}^1$.

Fig. 1: Illustration of a packing 3-coloring for $\mathbb{Z}^1$.

While Example 1 shows that $\chi_\rho(\mathbb{Z}^1) = 3$, the question of computing $\chi_\rho(\mathbb{Z}^2)$, where $\mathbb{Z}^2$ is the graph with vertex set $\mathbb{Z} \times \mathbb{Z}$ and edges between orthogonally adjacent points (i.e., points whose ℓ_1 distance equals 1), has been open since the introduction of packing colorings by Goddard et al. [10]. On the other hand, it is known that $\chi_\rho(\mathbb{Z}^3) = \infty$ (again considering edges between points whose ℓ_1 distance equals 1) [9]. The problem of computing $3 \leq \chi_\rho(\mathbb{Z}^2) \leq \infty$ has received significant attention, and it is described as "*the most attractive [of the packing coloring problems over infinite graphs]*" by Brešar et al. [3]. We can now state our main theorem, providing a final answer to this problem.

Theorem 1. $\chi_\rho(\mathbb{Z}^2) = 15$.

An upper bound of 15 had already been proved by Martin et al. [18], who found a packing 15-coloring of a 72×72 grid that can be used for periodically tiling the entirety of $\mathbb{Z}^2$. Therefore, the main contribution of our work consists of proving that 14 colors are not enough for $\mathbb{Z}^2$. Table 1 presents a summary of the historical progress on computing $\chi_\rho(\mathbb{Z}^2)$. It is worth noting that amongst the computer-generated proofs (i.e., all since Soukal and Holub [22] in 2010), ours is the first one to be formally verified, see Section 4.

Table 1: Historical summary of the bounds known for $\chi_\rho(\mathbb{Z}^2)$.

Year	Citation	Approach	Lower bound	Upper bound
2002	Goddard et al. [10]	Manual	9	23
2002	Schwenk [20]	Unkown	9	22
2009	Fiala et al. [8]	Manual + Computer	10	23
2010	Soukal and Holub [22]	Simulated Annealing	10	17
2010	Ekstein et al. [7]	Brute Force Program	12	17
2015	Martin et al. [17]	SAT solver	13	16
2017	Martin et al. [18]	SAT solver	13	15
2022	Subercaseaux and Heule [23]	SAT solver	14	15
2022	**This article**	SAT solver	15	15

For any $k \geq 4$, the problem of determining whether a graph G admits a packing 4-coloring is known to be NP-hard [10], and thus we do not expect a polynomial time algorithm for computing $\chi_\rho(\cdot)$. This naturally motivates the use of satisfiability (SAT) solvers for studying the packing chromatic number of finite subgraphs of $\mathbb{Z}^2$. The rest of this article is thus devoted to proving Theorem 1 by using automated reasoning techniques, in a way that produces a proof that can be checked independently and that has been checked by verified software.

2 Background

We start by recapitulating the components used to obtain a lower bound of 14 in our previous work [23]. Naturally, in order to prove a lower bound for $\mathbb{Z}^2$ one needs to prove a lower bound for a finite subgraph of it. As in earlier work, we consider *disks* (i.e., 2-dimensional balls in the ℓ_1-metric) as the finite subgraphs to study [23]. Concretely, let $D_r(v)$ be the subgraph induced by $\{u \in V(\mathbb{Z}^2) \mid d(u,v) \leq r\}$. To simplify notation, we use D_r as a shorthand for $D_r((0,0))$, and we let $D_{r,k}$ be the instance consisting of deciding whether D_r admits a packing k-coloring. Moreover, let $D_{r,k,c}$ be the instance $D_{r,k}$ but enforcing that the central vertex $(0,0)$ receives color c (Fig. 2).

For example, a simple lemma of Subercaseaux and Heule [23, Proposition 5] proves that the unsatisfiability of $D_{3,6,3}$ is enough to deduce that $\chi_\rho(\mathbb{Z}^2) \geq 7$. We will prove a slight variation of it (Lemma 2) later on in order to prove Theorem 1, but for now let us summarize how they proved that $D_{12,13,12}$ is unsatisfiable.

Encodings. The direct encoding for $D_{r,k,c}$ consists simply of variables $x_{v,t}$ stating that vertex v gets color t, as well as the following clauses:

1. (at-least-one-color clauses, ALOC) $\bigvee_{t=1}^{k} x_{v,t}, \quad \forall v \in V,$
2. (at-most-one-distance clauses, AMOD)

$$\overline{x_{u,t}} \vee \overline{x_{v,t}}, \quad \forall t \in \{1,\ldots,k\}, \forall u,v \in V \text{ s.t. } 0 < d(u,v) \leq t,$$

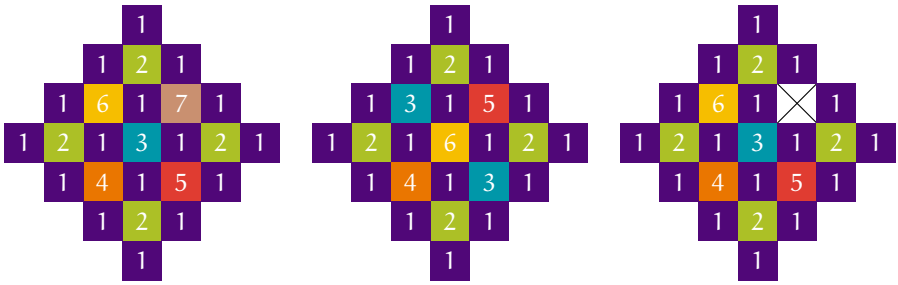


Fig. 2: Illustration of satisfying assignments for $D_{3,7,3}$ and $D_{3,6,6}$. On the other hand, $D_{3,6,3}$ is not satisfiable.

3. (center clause) $x_{(0,0),c}$.

This amounts to $O(r^2k^3)$ clauses [23]. The recursive encoding is significantly more involved, but it leads to only $O(r^2k \log k)$ clauses asymptotically. Unfortunately, the constant involved in the asymptotic expression is large, and this encoding did not give them practical speed-ups [23].

Cube And Conquer. Introduced by Heule et al. [13], the *Cube And Conquer*approach aims to *split* a SAT instance φ into multiple SAT instances $\varphi_1, \ldots, \varphi_m$ in such a way that φ is satisfiable if, and only if, at least one of the instances φ_i is satisfiable; thus allowing to work on the different instances φ_i in parallel. If $\psi = (c_1 \vee c_2 \vee \cdots \vee c_m)$ is a tautological DNF, then we have

$$\text{SAT}(\varphi) \iff \text{SAT}(\varphi \wedge \psi) \iff \text{SAT}\left(\bigvee_{i=1}^{m} (\varphi \wedge c_i)\right) \iff \text{SAT}\left(\bigvee_{i=1}^{m} \varphi_i\right),$$

where the different $\varphi_i \coloneqq (\varphi \wedge c_i)$ are the instances resulting from the split.

Intuitively, each cube c_i represents a *case*, i.e., an assumption about a satisfying assignment to φ, and soundness comes from ψ being a tautology, which means that the split into cases is exhaustive. If the split is well designed, then each φ_i is a particular case that is substantially easier to solve than φ, and thus solving them all in parallel can give significant speed-ups, especially considering the sequential nature of CDCL, at the core of most solvers. Our previous work [23] proposed a concrete algorithm to generate a split, which already results in an almost linear speed-up, meaning that by using 128 cores, the performance gain is roughly a $\times 60$ factor.

Symmetry Breaking. The idea of *symmetry breaking* [6] consists of exploiting the symmetries that are present in SAT instances to speed-up computation. In particular, $D_{r,k,c}$ instances have 3 axes of symmetry (i.e., vertical, horizontal, and diagonal) which allowed for close to an 8-fold improvement in performance for proving $D_{12,13,12}$ to be unsatisfiable. The particular use of symmetry breaking in our previous approach [23] was happening at the *Cube And Conquer* level, where

out of the sub-instances $\varphi_i, \ldots, \varphi_m$ produced by the split, only a $1/8$-fraction of them had to be solved, as the rest were equivalent under isomorphism.

Verification. Arguably the biggest drawback of our previous approach proving a lower bound of 14 is that it lacked the capability of generating a computer-checkable proof. To claim a full solution to the 20-year-old problem of computing $\chi_\rho(\mathbb{Z}^2)$ that is accepted by the mathematics community, we deem paramount a fully verifiable proof that can be scrutinized independently.

The most commonly-used proofs for SAT problems are expressed in the DRAT clausal proof system [11]. A DRAT proof of unsatisfiability is a list of clause addition and clause deletion steps. Formally, a clausal proof is a list of pairs $\langle \mathsf{s}_1, C_1 \rangle, \ldots, \langle \mathsf{s}_m, C_m \rangle$, where for each $i \in 1, \ldots, m$, $s_i \in \{\mathsf{a}, \mathsf{d}\}$ and C_i is a clause. If $s_i = \mathsf{a}$, the pair is called an *addition*, and if $s_i = \mathsf{d}$, it is called a *deletion*. For a given input formula φ_0, a clausal proof gives rise to a set of *accumulated formulas* φ_i ($i \in \{1, \ldots, m\}$) as follows:

$$\varphi_i = \begin{cases} \varphi_{i-1} \cup \{C_i\} & \text{if } \mathsf{s}_i = \mathsf{a} \\ \varphi_{i-1} \setminus \{C_i\} & \text{if } \mathsf{s}_i = \mathsf{d} \end{cases}$$

Each clause addition must preserve satisfiability, which is usually guaranteed by requiring the added clauses to fulfill some efficiently decidable syntactic criterion. The main purpose of deletions is to speed up proof checking by keeping the accumulated formula small. A valid proof of unsatisfiability must end with the addition of the empty clause.

3 Optimizations

Even with the best choice of parameters for our previous approach, solving the instance $D_{12,13,12}$ takes almost two days of computation with a 128-core machine [23]. In order to prove Theorem 1, we will require to solve an instance roughly 100 times harder, and thus several optimizations will be needed. In fact, we improve on all aspects discussed in Section 2; we present five different forms of optimization that are key to the success of our approach, which we summarize next.

1. We present a new encoding, which we call the plus *encoding* that has conceptual similarities with the recursive encoding of Subercaseaux and Heule [23], while achieving a significant gain in practical efficiency.
2. We present a new split algorithm that works substantially better than the previous split algorithm when coupled with the plus encoding.
3. We improve on symmetry breaking by using multiple layers of symmetry-breaking clauses in a way that exploits the design of the split algorithm to increase performance.
4. We study the choice of color to fix at the center, showing that one can gain significantly in performance by making instance-based choices; for example, $D_{12,13,6}$ can be solved more than three times as fast as $D_{12,13,12}$ (the instance used by Subercaseaux and Heule [23]).

5. We introduce a new and extremely simple kind of clauses called ALOD clauses, which improve performance when added to the other clauses of any encoding we have tested.

The following subsections present each of these components in detail.

3.1 "*Plus*": a New Encoding

Despite the asymptotic improvement of the recursive encoding of Subercaseaux and Heule [23], its contribution is mostly of "theoretical interest" as it does not improve solution times. Nonetheless, that encoding suggests the possibility of finding one that is both more succinct than the direct encoding and that speed-ups computation. Our path towards such an encoding starts with *Bounded Variable Addition (BVA)* [16], a technique to automatically re-encode CNF formulas by adding new variables, with the goal of minimizing their resulting size (measured as the sum of the number of variables and the number of clauses). BVA can significantly reduce the size of $D_{r,k,c}$ instances, even further than the recursive encoding. Moreover, BVA actually speeds-up computation when solving the resulting instances with a CDCL solver, see Table 2. Figure 3 compares the number of AMOD clauses between the direct encoding and the BVA encoding; for example in the direct encoding, for D_{14} color 10 would require roughly 30000 clauses, whereas it requires roughly 3500 in the BVA encoding. It can be observed as well in Figure 3 that the direct encoding grows in a very structured and predictable way, where color c in D_r requires roughly r^2c^2 clauses. On the other hand, arguably because of its locally greedy nature, the results for BVA are far more erratic, and roughly follow a $4r^2 \lg c$ curve.

The encoding resulting from BVA does not perform particularly well when coupled with the split algorithm of Subercaseaux and Heule. Indeed, Table 2 shows that while BVA heavily improves runtime under sequential CDCL, it does not provide a meaningful advantage when using *Cube And Conquer*. Furthermore, encodings resulting from BVA are hardly interpretable, as BVA uses

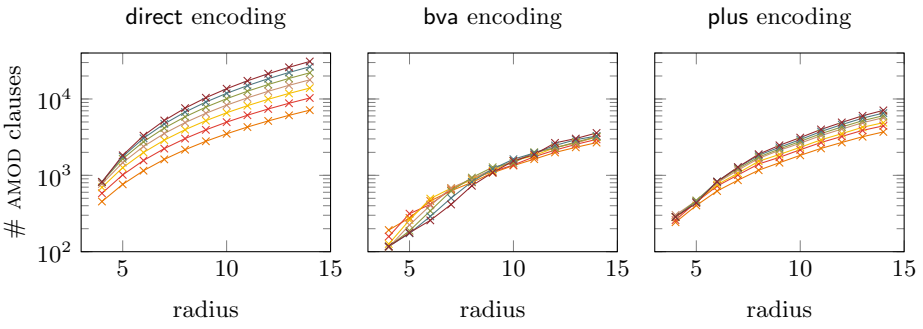


Fig. 3: Comparison of the size of the at-most-one-color clauses between the direct encoding and the BVA-encoding, for D_4 up to D_{14} and colors $\{4, \ldots, 10\}$.

Table 2: Comparison between the different encodings. *Cube And Conquer* experiments were performed with the approach of Subercaseaux and Heule [23] (parameters $F = 5, d = 2$) on a 128-core machine. Hardware details in Section 5.

	direct encoding		bva encoding		plus encoding	
	$D_{5,10,5}$	$D_{6,11,6}$	$D_{5,10,5}$	$D_{6,11,6}$	$D_{5,10,5}$	$D_{6,11,6}$
Number of variables	610	935	973	1559	673	1039
Number of clauses	10688	21086	2313	3928	4063	7548
CDCL runtime (s)	255.12	10774.79	39.88	2539.38	15.90	811.66
Cube-and-conquer wall-clock (s)	0.77	26.20	0.78	17.97	0.50	6.68

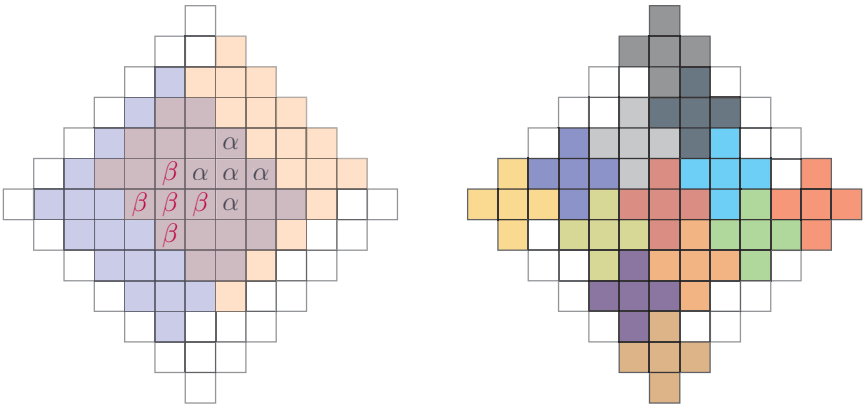
a locally greedy strategy for introducing new variables. As a result, the design of a split algorithm that could work well with BVA is a very complicated task. Therefore, our approach consisted of reverse engineering what BVA was doing over some example instances, and using that insight to design a new encoding that produces instances of size comparable to those generated by BVA while being easily interpretable and thus compatible with natural split algorithms.

By manually inspecting BVA encodings one can deduce that a fundamental part of their structure is what we call *regional variables/clauses*. A regional variable $r_{S,c}$ is associated with a set of vertices S and a color c, meaning that at least one vertex in S receives color c. Let us illustrate their use with an example.

Example 2. Consider the instance $D_{6,11}$, and let us focus on the *at-most-one-distance* (AMOD) clauses for color 4. Figure 4a depicts two regional clauses: one in orange (vertices labeled with α), and one in blue (vertices labeled with β), each consisting of 5 vertices organized in a *plus* (+) shape. We thus introduce variables $r_{\text{orange},4}$ and $r_{\text{blue},4}$, defined by the following clauses:

1. $\overline{r_{\text{orange},4}} \vee \bigvee_{v \text{ has label } \alpha} x_{v,4}$,
2. $\overline{r_{\text{blue},4}} \vee \bigvee_{v \text{ has label } \beta} x_{v,4}$,
3. $r_{\text{orange},4} \vee \overline{x_{v,4}}$, for each v with label α,
4. $r_{\text{blue},4} \vee \overline{x_{v,4}}$, for each v with label β.

The benefit of introducing these two new variables and $2 + (5 \cdot 2) = 12$ additional clauses will be shown now, when using them to forbid conflicts more compactly. Indeed, each vertex labeled with α or β participates in $|D_4| - 1 = 40$ AMOD clauses in the direct encoding, which equals a total of $10 \cdot 40 - \binom{10}{2} = 355$ clauses for all of them (subtracting the clauses counted twice). However, note that all 36 vertices shaded in light orange are at distance at most 4 from all vertices labeled with α, and thus they are in conflict with $r_{\text{orange},4}$. This means that we can encode all conflicts between α-vertices and orange-shaded vertices with 36 clauses. The same can be done for β-vertices and the 36 vertices shaded in light blue. Moreover, all pairs of vertices (x, y) with x being an α-vertex and y being a β-vertex are in conflict, which we can represent simply with the clause $(r_{\text{orange},4} \vee r_{\text{blue},4})$, instead of $5 \cdot 5 = 25$ pairwise clauses. We still need,



(a) Illustration of regions interacting in $P_{6,11,6}$, for color 4.

(b) Illustration of the placement of regions of the 13 regions in $P_{6,11,6}$.

Fig. 4: Illustrations for $P_{6,11,6}$.

however, to forbid that more than one α-vertex receives color 4, and the same for β-vertices, which can be done by simply adding all $2 \cdot \binom{5}{2} = 20$ AMOD clauses between all pairs. In total, the total number of clauses involving α or β vertices has gone down to $12+2\cdot 36+20+1 = 105$ clauses, from the original 355 clauses, by merely adding two new variables.

As shown in Example 2, the use of regional clauses can make encodings more compact, and this same idea scales even better for larger instances when the regions are larger. A key challenge for designing a *regional encoding* in this manner is that it requires a choice of regions (which can even be different for every color). After trying several different strategies for defining regions, we found one that works particularly well in practice (despite not yielding an optimal number for the metric #variables + #clauses), which we denote the plus *encoding*. The plus encoding is based on simply using "+" shaped regions (i.e., D_1) for all colors greater than 3, and to not introduce any changes for colors 1, 2 and 3 as they only amount to a very small fraction of the total size of the instances we consider. We denote with $P_{d,k,c}$ the plus encoding of the diamond of size d with k colors, and the centered being colored with c. Figure 4b illustrates $P_{6,11,6}$. Interestingly, the BVA encoding opted for larger regions for the larger colors, using for example D_2's or D_3's as regions for color 14. We have experimentally found this to be very ineffective when coupled with our split algorithms. In terms of the locations of the "+" shaped regions, we have placed them manually through an interactive program, arriving to the conclusion that the best choice of locations consists of packing as many regions as possible and as densely around the center as possible. A more formal presentation of all the clauses involved in the plus encoding is presented in the extended *arXiv* version [24] of this paper, but all its components have been illustrated in Example 2.

The exact number of clauses resulting from the plus encoding is hard to analyze precisely, but it is clear that asymptotically it only improves from the direct encoding by a constant multiplicative factor. Figure 3 and Table 2 illustrate the compactness of the plus encoding over particular instances, and its increase in efficiency both for CDCL solving as well as with the *Cube And Conquer* approach of Subercaseaux and Heule [23].

3.2 Symmetry Breaking

Another improvement of our approach is a static symmetry-breaking technique, while Subercaseaux and Heule [23] achieved symmetry breaking by discarding all but 1/8 of the cubes. We cannot do this easily since the plus encoding does not have an 8-fold symmetry. Instead it has a 4-fold symmetry (see Figure 4b). We add symmetry breaking clauses directly on top of the direct encoding (i.e., instead of using it after a *Cube And Conquer* split), as $D_{r,k,c}$ has indeed an 8-fold symmetry (see Figure 5b). Concretely, if we consider a color t, it can only appear once in the $D_{\lfloor t/2 \rfloor}$, as if it appeared more than once said appearances would be at distance $\leq t$. Given this, we can assume without loss of generality that if there is one appearance of t in $D_{\lfloor t/2 \rfloor}$, then it appears with coordinates (a, b) such that $a \geq 0 \wedge b \geq a$. We enforce this by adding negative units of the form $\overline{x_{(i,j),t}}$ for every pair $(i, j) \in D_{\lfloor t/2 \rfloor}$ such that $i < 0 \vee j < i$. This is illustrated in Figure 5b for $D_{5,10}$. Note however that this can only be applied to a single color t, as when a vertex in the *north-north-east* octant gets assigned color t, the 8-fold symmetry is broken. However, if the symmetry breaking clauses have been added for color t, and yet t does not appear in $D_{\lfloor t/2 \rfloor}$, then there is still an 8-fold symmetry in the encoding we can exploit by breaking symmetry on some other color t'. This way, our encoding uses $L = 5$ *layers* of symmetry breaking, for colors $k, k-1, \ldots, k-L+1$. At each layer i, where symmetry breaking is done over color $k - i$, except for the first (i.e., $i > 0$), we need to concatenate a clause

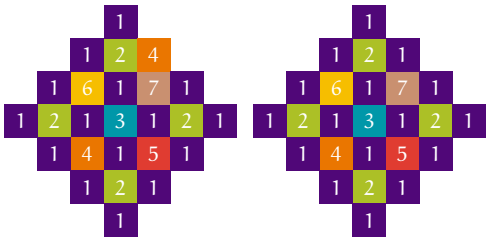
$$\text{SymmetryBroken}_i \coloneqq \bigvee_{t=k-i}^{k} \bigvee_{\substack{(a,b) \in D_{\lfloor t/2 \rfloor} \\ 0 \leq a \leq b}} x_{(a,b),t}$$

to each symmetry breaking clause, so that symmetry breaking is applied only when symmetry has not been broken already. Table 3 (page 14) illustrates the impact of this symmetry breaking approach, yielding close to a $\times 40$ speed-up for $D_{6,11,6}$.

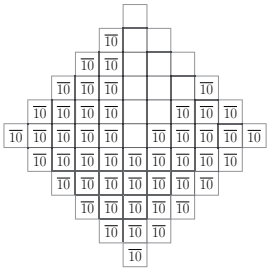
3.3 *At-Least-One-Distance* clauses

Yet another addition to our encoding is what we call *At-Least-One-Distance* (ALOD) clauses, which consist on stating that, for every vertex v, if we consider $D_1(v)$, then at least one vertex in $D_1(v)$ must get color 1. Concretely, the *At-Least-One-Distance* clause corresponding to a vertex $v = (i, j)$ is

$$C_v = x_{(i,j),1} \vee x_{(i+1,j),1} \vee x_{(i-1,j),1} \vee x_{(i,j+1),1} \vee x_{(i,j-1),1}.$$



(a) Illustration of the effect of adding ALOD clauses. The right figure, with ALOD clauses, presents a *chessboard pattern*.



(b) Some symmetry-breaking unit clauses added to $D_{5,10}$.

Fig. 5: The effect of adding ALOD clauses (left) and symmetry-breaking (right).

Note that adding these clauses preserves satisfiability since they are *blocked clauses* [15]; this can be seen as follows. If no vertex in $D_1(v)$ gets assigned color 1, then we can simply assign $x_{v,1}$, thus satisfying the new clause C_v.

The purpose of ALOD clauses can be described as *incentives* towards assigning color 1 in a *chessboard pattern* (see Figure 5a), which seems to simplify the rest of the computation. Empirically, their addition improves runtimes; see Table 3.

3.4 Cube And Conquer Using Auxiliary Variables

The split of Subercaseaux and Heule [23] is based on cases about the $x_{v,c}$ variables of the direct encoding, and specifically using vertices v that are close to the center and colors c that are in the top-t colors for some parameter t.

Our algorithm is instead based on cases only around the new regional variables $r_{S,c}$, which appears to be key for exploiting their use in the encoding.

More concretely, our algorithm, which we call PTR, is roughly based on splitting the instance into cases according to which out of the R regions that are closest to the center get which of the T highest colors (noting that a region can get multiple colors). A third parameter P indicates the maximum number of positive literals in any cube of the split. More precisely, there are cubes with i positive literals for $i \in \{0, 1, \ldots, P-1, P\}$, and the set of cubes with i positive literals is constructed by PTR as follows:

1. Let $\mathcal{R}$ be the set of R regions that are the closest to the center, and $\mathcal{T}$ the set consisting of the T highest colors (i.e., $\{k, k-1, \ldots, k-T+1\}$).
2. For each of the R^i tuples $\vec{S} \in \mathcal{R}^i$, we create $\binom{T}{i}$ cubes as described in the next step.
3. For each subset $Q \subseteq \mathcal{T}$ with size $|Q| = i$, let $q_1, \ldots, q_i$ be its elements in increasing order, and then create a cube with positive literals $r_{\vec{S}_j, q_j}$ for $j \in \{1, \ldots, i\}$. Then, if $i < P$, add to the cube negative literals $\overline{r_{\vec{S}_j, q_\ell}}$ for $j \in \{1, \ldots, i\}$ and every $q_\ell \notin Q$.

Lemma 1. *The cubes generated by the* PTR *algorithm form a tautology.*

The proof of Lemma 1 is quite simple, and we refer the reader to the proof of Lemma 7 in Subercaseaux and Heule [23] for a very similar one. Moreover, because our goal is to have a verifiable proof, instead of relying on Lemma 1, we test explicitly that the cubes generated by our algorithm form a tautology in all the instances mentioned in this paper. Pseudo-code for PTR is presented in the extended *arXiv* version of this paper [24].

3.5 Optimizing the Center Color

Our previous work [23] argued that for an instance $D_{r,k}$, one should fix the color of the central vertex to $\min(r, k)$. However, our experiments suggest otherwise. As the proof of Lemma 2 (in extended *arXiv* version [24]) implies, we are allowed to fix any color in the center, and as long as the resulting instance is unsatisfiable, that will allow us to establish the same lower bound. It turns out that the choice of the center color can dramatically affect performance, as shown for instance $D_{12,13}$ (the one used to prove $\chi_\rho(\mathbb{Z}^2) \geq 14$) in Figure 6. Interestingly, performance does not change monotonically with the value fixed in the center. Intuitively, it appears that fixing smaller colors in the center is ineffective as they impose restrictions on a small region around the center, while fixing very large colors in the center does not constrain the center much; for example, on the one hand, fixing a 1 or 2 in the center does not seem to impose any serious constraints on solutions. On the other hand, when a 12 is fixed in the center (as in our previous work [23]), color 6 can be used 5 times in D_6, whereas if color 6 is fixed in the center, it can only be used once in D_6. The apparent advantage of fixing 12 in the center (that it cannot occur anywhere else in $D_{12,13}$), is outweighed by the extra constraints around the center that fixing color 6 imposes; Subercaseaux and Heule already observed that most conflicts between colors occur around the center [23]), thus explaining why it makes sense to optimize in that area.

The main result of Subercaseaux and Heule [23] is the unsatisfiability of $D_{12,13,12}$, which required 45 CPU hours using the same SAT solver and similar hardware. Let $P^\star_{d,k,c}$denote $P_{d,k,c}$ with ALOD clauses and symmetry-breaking

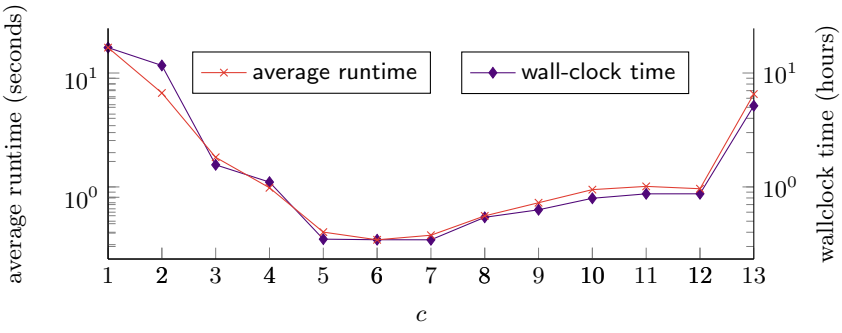


Fig. 6: The impact of the color in the center (c) on the performance for $P^\star_{12,13,c}$.

$$\overbrace{D_{15,14,6} \equiv \underbrace{D^{\star}_{15,14,6}}_{\text{re-encoding proof}}}^{\text{symmetry proof}} \equiv \overbrace{P^{\star}_{15,14,6} \vDash \underbrace{N_{15,14,6} \vDash \bot}_{\text{tautology proof}}}^{\text{implication proof}}$$

Fig. 7: Illustration of the verification pipeline.

predicates. We show unsatisfiability of $P^{\star}_{12,13,12}$ in 1.18 CPU hours and of $P^{\star}_{12,13,6}$ in 0.34 CPU hours. So the combination of the plus encoding and the improved center reduces the computational costs by two orders of magnitude.

4 Verification

Our pipeline proves that, in order to trust $\chi_\rho(\mathbb{Z}^2) = 15$ as a result, the only component that requires unverified trust is the direct encoding of $D_{15,14,6}$. Indeed, let $P^{\star}_{15,14,6}$ be the instance $P_{15,14,6}$ with ALOD-clauses and 5 layers of symmetry breaking clauses, and let $\psi = \{c_1, \ldots, c_m\}$ be the set of cubes generated by the PTR algorithm with parameters $P = 6, T = 7, R = 9$. We then prove:

1. that $D_{15,14,6}$ is satisfiability equivalent to $P^{\star}_{15,14,6}$.
2. the DNF $\psi = c_1 \vee c_2 \vee \cdots \vee c_m$ is a tautology.
3. each instance $(P^{\star}_{15,14,6} \wedge c_i)$, for $c_i \in \psi$ is unsatisfiable.
4. hence the negation of each cube is implied by $P^{\star}_{15,14,6}$.
5. since ψ is a tautology, its negation $N_{15,14,6}$ is unsatisfiable.

As a result, Theorem 1 relies only on our implementation of $D_{15,14,6}$. Fortunately, this is quite simple, and the whole implementation is presented in the extended *arXiv* version of this paper [24]. Figure 7 illustrates the verification pipeline, and the following paragraphs detail its different components.

Symmetry Proof. The first part of the proof consists in the addition of symmetry-breaking predicates to the formula. This part needs to go before the re-encoding proof, because the plus encoding does not have the 8-fold symmetry of the direct encoding. Each of the clauses in the symmetry-breaking predicates have the substitution redundancy (SR) property [5]. This is a very strong redundancy property and checking whether a clause C has SR w.r.t. a formula φ is NP-complete. However, since we know the symmetry, it is easy to compute a SR certificate. There exists no SR proof checker. Instead, we implemented a prototype tool to convert SR proofs into DRAT for which formally verified checkers exists. Our conversion is similar to the approach to converted propagation redundancy into DRAT [12]. The conversion can significantly increase the size of the proof, but the other proof parts are typically larger for harder formulas, thus the size is acceptable.

Re-encoding Proof. After symmetry breaking, the formula encoding is optimized by transforming the direct encoding into the plus encoding and adding the ALOD clauses. This part of the proof is easy. All clauses in the plus encoding and all ALOD clauses have the RAT redundancy property w.r.t. the direct encoding. This means that we can add all these clauses with a single addition step per clause. Afterward, the clauses that occur in the direct encoding but not in the plus encoding are removed using deletion steps.

Implication Proof. The third part of the proof expresses that the formula cannot be satisfied with any of the cubes from the split. For easy problems, one can avoid splitting and just use the empty cube as tautological DNF. For harder problems, splitting is crucial. We solve $D_{15,14,6}$ using a split with just over 5 million cubes. Using a SAT solver to show that the formula with a cube is unsatisfiable shows that the negative of the cube is implied by the formula. We can derive all these implied clauses in parallel. The proofs of unsatisfiability can be merged into a single implication proof.

Tautology Proof. The final proof part needs to show that the negation of the clauses derived in the prior steps form a tautology. In most cases, including ours, the cubes are constructed using a tree-based method. This makes the tautology check easy as there exists a resolution proof from the derived clauses to the empty clause using $m-1$ resolution steps with m denoting the number of cubes. This part can be generated using a simple SAT call.

The final proof merges all the proof parts. In case the proof parts are all in the DRAT format, such as our proof parts, then they can simply be merged by concatenating the proofs using the order presented above.

5 Experiments

Experimental Setup. In terms of hardware, all our experiments were run in the Bridges2 [4] supercomputer. Each node has the following specifications: Two AMD EPYC 7742 CPUs, each with 64 cores, 256MB of L3 cache, and 512GB total RAM memory. Our code and various formulas are publicly available at the repository `https://github.com/bsubercaseaux/PackingChromaticTacas`. In terms of software, all sequential experiments were run on state-of-the-art solver `CaDiCaL` [2], while parallel experiments with *Cube And Conquer* were run using a new implementation of parallel `iCaDiCaL` because it supports incremental solving [13] while being significantly faster than `iLingeling`.

Effectiveness of the Optimizations. We evaluated the optimizations to the direct encoding as proposed in Section 3: the plus encoding, the addition of the ALOD clauses, and the new symmetry breaking. The results are shown in Table 3. We picked $D_{6,11,6}$ for this evaluation since it is the largest diamond that can still be solved within a couple of hours on a single core.

The main conclusion is that the optimizations significantly improve the runtime. A comparison between the direct encoding without symmetry breaking and

the plus encoding with symmetry breaking and the ALOD clauses shows that the latter can be solved roughly 200x faster. Table 3 shows all 8 possible configurations. Turning on any of the optimizations always improves performance. The effectiveness of the plus encoding and ALOD clauses is somewhat surprising: the speed-up factor obtained by re-encoding typically does not exceed the factor by which the formula size is reduced. In this case, the reduction factor in formula size is less than 3, while the speed-up is larger than 13 (see the difference between the first and second row of Table 3). Moreover, we are not aware of the effectiveness of adding blocked clauses. Typically SAT solvers remove them.

We also constructed DRAT proofs of the optimizations (shown as derivation in the table) and the solver runtime. We merged them into a single DRAT proof by concatenating the files. The proofs were first checked with the `drat-trim` tool, which produced LRAT proofs. These LRAT files were validated using the formally-verified `cake-lpr` checker. The size of the DRAT proofs and the checking time are shown in the table. Note that the checking time for the proofs with symmetry breaking is always larger than the solving times. This is caused by expressing the symmetry breaking in DRAT resulting in a 436 Mb proof part.

The Implication Proof. The largest part of the computation consist of showing that $P^{\star}_{15,4,6}$ is unsatisfiable under each of the $5,217,031$ cubes produced by the cube generator. The results of the experiments are shown in Figure 8 (left). The left plot shows that roughly half of the cubes can be solved in a second or less. The average runtime of cubes was 3.35 seconds, while the hardest cube required 1584.61 seconds. The total runtime was 4851.38 CPU hours.

For each cube, we produced a compressed DRAT proof (the default output of `CaDiCaL`). Due to the lack of hints in DRAT proofs, they are somewhat complex to validate using a formally-verified checker. Instead, we use the tool `drat-trim` to trim the proofs and add hints. The result are uncompressed LRAT files, which we validate using the formally-verified checker `cake_lpr`. The verification time was 4336.93 CPU hours, so slightly less than the total runtime.

The sizes of each of the implication proofs show a similar distribution, as depicted in Figure 8 (right). Most proofs are less than 10 MB in size. The

Table 3: Evaluating the effectiveness of the optimizations on $D_{6,11,6}$.

sym	ALOD	plus	#var	#cls	runtime	derivation	proof	check
			935	21086	10741.69	0 b	11.99 Gb	31731.20
		x	1039	7548	809.65	149 Kb	1.29 Gb	1720.82
	x		935	21171	8422.38	1.6 Kb	8.11 Gb	21732.74
	x	x	1039	7633	389.71	151 Kb	1.29 Gb	1708.21
x			935	21286	273.19	436 Mb	0.63 Gb	1390.04
x		x	1039	7748	66.74	436 Mb	0.14 Gb	1022.42
x	x		935	21371	252.71	436 Mb	0.68 Gb	1359.05
x	x	x	1039	7833	55.56	436 Mb	0.10 Gb	997.90

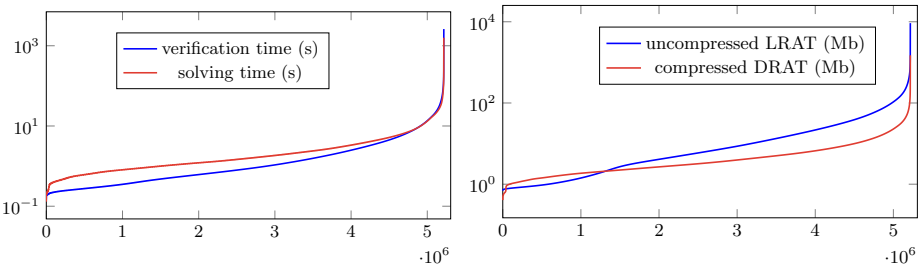


Fig. 8: Cactus plot of solving and verification times in seconds (left) and cactus plot of the size of the compressed DRAT proof and uncompressed LRAT proof in Mb (right).

compressed DRAT proofs are generally smaller compared to the LRAT proofs, but that is mostly due to compression, which reduces the size by around 70%.

The Chessboard Conjecture and its Counterexample. Given that color 1 can be used to fill in $1/2$ of $\mathbb{Z}^2$ in a packing coloring, and the packing colorings found in the past, with $15, 16$ or 17 colors used color 1 with density $1/2$ in a *chessboard pattern* [18], it is tempting to assume that this must always be the case. This way, we conjectured that any instance $D_{r,k,c}$ is satisfiable if and only if it is with the chessboard pattern. The consequence of the conjecture is significant, as if it were true we could fix half of the vertices to color 1, thus massively reducing the size of the instance and its runtime. Unfortunately, this conjecture happens to be false, with the smallest counterexample being $D_{14,14,6}$ as illustrated in Figure 9, which deviates from the chessboard pattern in only 2 vertices. We have proved as well that no solution for $D_{14,14,6}$ deviating in only 1 vertex from the chessboard pattern exists.

Proving the Lower Bound. In order to prove Theorem 1, we require the following 3 lemmas, from where the conclusion easily follows.

Lemma 2. *If $D_{15,14,6}$ is unsatisfiable, then $\chi_\rho(\mathbb{Z}^2) \geq 15$.*

Lemma 3. *If $D_{15,14,6}$ is satisfiable, then $P^\star_{15,14,6}$ is also satisfiable.*

Lemma 4. *$P^\star_{15,14,6}$ is unsatisfiable.*

We have obtained computational proofs of Lemma 3 and Lemma 4 as described above, and thus it only remains to prove Lemma 2, which we include in the appendix. We can thus proceed to our main proof.

Proof (of Theorem 1). Since Martin et al. proved that $\chi_\rho(\mathbb{Z}^2) \leq 15$ [18], it remains to show $\chi_\rho(\mathbb{Z}^2) \geq 15$, which by Lemma 2 reduces to proving Lemma 3 and Lemma 4. We have proved these lemmas computationally, obtaining a single DRAT proof as described in Section 4. The total solving time was 4851.31 CPU hours, while the total checking time of the proofs was 4336.93 CPU hours. The total size of the compressed DRAT proof is 34 terabytes, while the uncompressed LRAT proof weighs 122 terabytes.

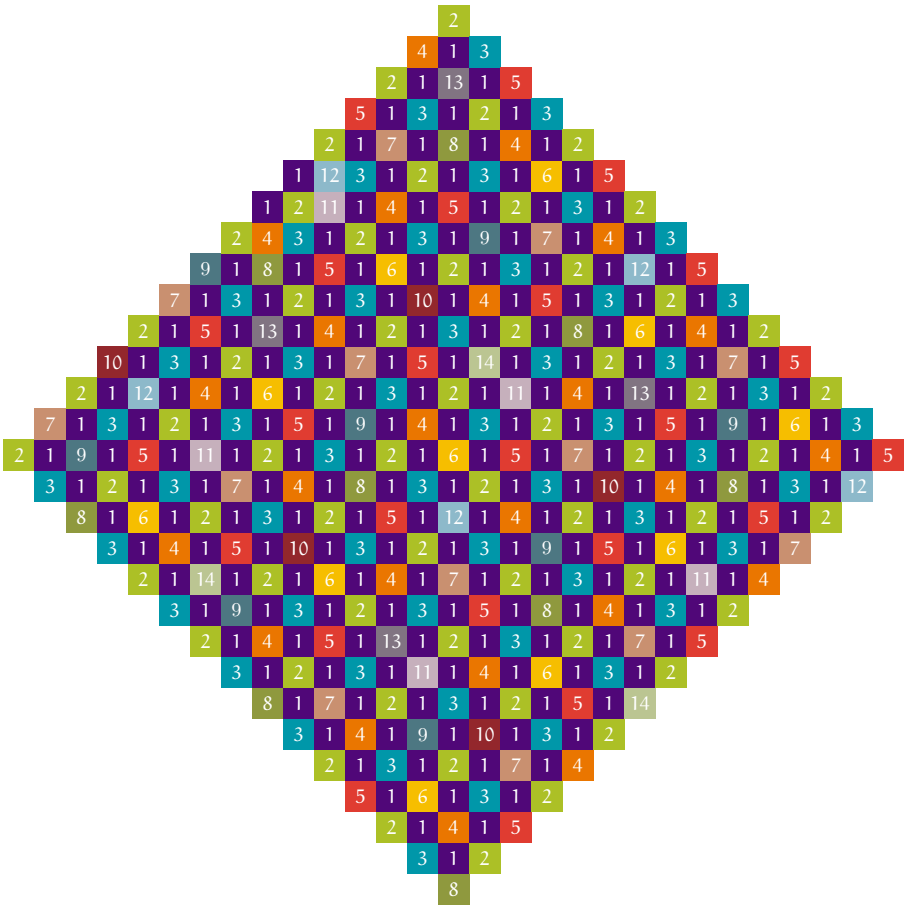


Fig. 9: A valid coloring of $D_{14,14,6}$. No valid coloring exists for this grid with a full chessboard pattern of 1's.

6 Concluding Remarks and Future Work

We have proved $\chi_\rho(\mathbb{Z}^2) = 15$ by using several SAT-solving techniques, in what constitutes a new success story for automated reasoning tools applied to combinatorial problems. Moreover, we believe that several of our contributions in this work might be applicable to other settings and problems. Indeed, we have obtained a better encoding by reverse engineering BVA, and designed a split algorithm that works well coupled with the new encoding; this experience suggests the *split-encoding compatibility* as a new key variable to pay attention to when solving combinatorial problems under the *Cube And Conquer* paradigm. As for future work, it is natural to study whether our techniques can be used to improve other known bounds in the packing-coloring area (see e.g., [3]), as well as to other families of coloring problems, such as *distance colorings* [14].

Acknowledgements We thank the Pittsburgh Supercomputing Center for allowing us to use Bridges2 [4] in our experiments. We thank as well the anonymous reviewers for their comments and suggestions. We also thank Donald Knuth for his thorough comments and suggestions. The first author thanks the Facebook group *"actually good math problems"*, from where he first learned about this problem, and in particular to Dylan Pizzo for his post about this problem.

References

1. Appel, K., Haken, W.: Every planar map is four colorable. Part I: Discharging. Illinois Journal of Mathematics **21**(3), 429 – 490 (1977)
2. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
3. Brešar, B., Ferme, J., Klavžar, S., Rall, D.F.: A survey on packing colorings. Discussiones Mathematicae Graph Theory **40**(4), 923 (2020)
4. Brown, S.T., Buitrago, P., Hanna, E., Sanielevici, S., Scibek, R., Nystrom, N.A.: Bridges-2: A Platform for Rapidly-Evolving and Data Intensive Research, pp. 1–4. Association for Computing Machinery, New York, NY, USA (2021)
5. Buss, S., Thapen, N.: DRAT proofs, propagation redundancy, and extended resolution. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing – SAT 2019. pp. 71–89. Springer International Publishing, Cham (2019)
6. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: Proc. KR'96, 5th Int. Conf. on Knowledge Representation and Reasoning, pp. 148–159. Morgan Kaufmann (1996)
7. Ekstein, J., Fiala, J., Holub, P., Lidický, B.: The packing chromatic number of the square lattice is at least 12. CoRR **abs/1003.2291** (2010), http://arxiv.org/abs/1003.2291
8. Fiala, J., Klavžar, S., Lidický, B.: The packing chromatic number of infinite product graphs. Eur. J. Comb. **30**(5), 1101–1113 (jul 2009)
9. Finbow, A.S., Rall, D.F.: On the packing chromatic number of some lattices. Discrete Applied Mathematics **158**(12), 1224–1228 (2010), traces from LAGOS'07 IV Latin American Algorithms, Graphs, and Optimization Symposium Puerto Varas - 2007
10. Goddard, W., Hedetniemi, S., Hedetniemi, S., Harris, J., Rall, D.: Broadcast chromatic numbers of graphs. Ars Comb. **86** (01 2008)
11. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **abs/1610.06229** (2016), http://arxiv.org/abs/1610.06229
12. Heule, M.J.H., Biere, A.: What a difference a variable makes. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 75–92. Springer International Publishing, Cham (2018)
13. Heule, M.J.H., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Hardware and Software: Verification and Testing. pp. 50–65. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
14. Kramer, F., Kramer, H.: A survey on the distance-colouring of graphs. Discrete Mathematics **308**(2), 422–426 (2008)

15. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics **96-97**, 149–176 (1999)
16. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of boolean formulas. In: Proceedings of Haifa Verification Conference 2012 (2012)
17. Martin, B., Raimondi, F., Chen, T., Martin, J.: The packing chromatic number of the infinite square lattice is less than or equal to 16 (2015), http://arxiv.org/abs/1510.02374v1
18. Martin, B., Raimondi, F., Chen, T., Martin, J.: The packing chromatic number of the infinite square lattice is between 13 and 15. Discrete Applied Mathematics **225**, 136–142 (2017)
19. Neiman, D., Mackey, J., Heule, M.J.H.: Tighter bounds on directed Ramsey number R(7). Graphs and Combinatorics **38**(5), 156 (2022)
20. Schwenk, A.: private communication with Wayne Goddard. (2002)
21. Soifer, A.: The Hadwiger–Nelson Problem, pp. 439–457. Springer International Publishing, Cham (2016)
22. Soukal, R., Holub, P.: A note on packing chromatic number of the square lattice. The Electronic Journal of Combinatorics **17**(1), #N17 (Mar 2010)
23. Subercaseaux, B., Heule, M.J.H.: The Packing Chromatic Number of the Infinite Square Grid Is at Least 14. In: Meel, K.S., Strichman, O. (eds.) 25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, pp. 21:1–21:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022)
24. Subercaseaux, B., Heule, M.J.H.: The packing chromatic number of the infinite square grid is 15 (2023), https://arxiv.org/abs/2301.09757

Active Learning for SAT Solver Benchmarking

Tobias Fuchs(✉), Jakob Bach, and Markus Iser

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
info@tobiasfuchs.de, {jakob.bach,markus.iser}@kit.edu

Abstract. Benchmarking is a crucial phase when developing algorithms. This also applies to solvers for the SAT (propositional satisfiability) problem. Benchmark selection is about choosing representative problem instances that reliably discriminate solvers based on their runtime. In this paper, we present a dynamic benchmark selection approach based on active learning. Our approach predicts the rank of a new solver among its competitors with minimum runtime and maximum rank prediction accuracy. We evaluated this approach on the Anniversary Track dataset from the 2022 SAT Competition. Our selection approach can predict the rank of a new solver after about 10 % of the time it would take to run the solver on all instances of this dataset, with a prediction accuracy of about 92 %. We also discuss the importance of instance families in the selection process. Overall, our tool provides a reliable way for solver engineers to determine a new solver's performance efficiently.

Keywords: Propositional satisfiability · Benchmarking · Active learning

1 Introduction

One of the main phases of algorithm engineering is benchmarking. This also applies to propositional satisfiability (SAT), the archetypal $\mathcal{NP}$-complete problem. Benchmarking is, however, quite expensive regarding the runtime of experiments. While benchmarking a single SAT solver might still be feasible, developing new, competitive SAT solvers requires extensive experimentation with a variety of ideas [8,2]. In particular, a new solver idea is rarely best on the first try. Thus, it is highly desirable to reduce benchmarking time and discard unpromising ideas early, allowing to test more approaches or spend more time on promising ones. The field of SAT solver benchmarking is well established, but traditional benchmark selection approaches do not optimize benchmark runtime. Instead, they focus on selecting a representative set of instances for scoring solvers [10,15]. For the latter, SAT Competitions typically employ the PAR-2 score, i.e., the average runtime with a penalty of 2τ for timeouts with time-limit τ [8].

In this paper, we present a novel benchmark selection approach based on active learning. Our approach can predict the rank of a new solver with high accuracy in only a fraction of the time needed to evaluate the complete benchmark. Definition 1 specifies the problem we address.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 407–425, 2023
https://doi.org/10.1007/978-3-031-30823-9_21

Definition 1 (New-Solver Problem). *Given solvers $\mathcal{A}$, instances $\mathcal{I}$, runtimes $r\colon \mathcal{A}\times\mathcal{I}\to[0,\tau]$ with time-limit τ, and a new solver $\hat{a}\notin\mathcal{A}$, incrementally select benchmark instances from $\mathcal{I}$ to maximize the confidence in predicting the rank of $\hat{a}$ while minimizing the total benchmark runtime.*

Note that our scenario assumes knowing the runtimes of all solvers, except the new one, on all instances. One could also imagine a collaborative filtering scenario, where runtimes are only partially known [23,25].

Our approach satisfies several desirable criteria for benchmarking: Rather than outputting a binary classification, i.e., whether the new solver is worse than an existing solver or not, we provide a *scoring* function that shows by which margin a solver is worse and how similar it is to existing solvers. In particular, our approach enables *ranking* the new solver amidst a set of existing solvers. For this ranking, we do not even need to predict exact solver runtimes, which is trickier. Further, we optimize the *runtime* that our strategy needs to arrive at its conclusion. We use instance and runtime *features*. Moreover, we select instances *non-randomly* and *incrementally*. In particular, we consider runtime information from already done experiments when choosing the next. By doing so, we can control the properties of the benchmarking approach, such as its required runtime. Our approach is *scalable* in that it ranks a new solver $\hat{a}$ among any number of known solvers $\mathcal{A}$. In particular, we only subsample the benchmark once instead of comparing pairwise against each other solver [21].

We evaluate our approach with the SAT Competition 2022 Anniversary Track dataset [2], consisting of 5355 instances and runtimes of 28 solvers. We perform cross-validation by treating each solver once as the new solver and learning to predict the PAR-2 rank of that solver. On average, our predictions reach about 92 % accuracy with only about 10 % of the runtime required to evaluate these solvers on the complete set of instances.

Our entire source code[1] and experimental data[2] are available on GitHub.

2 Related Work

Benchmarking is not only of high interest in many fields but also an active research area on its own. Recent studies show that benchmark selection is challenging for multiple reasons. Biased benchmarks can easily lead to fallacious interpretations [7]. Benchmarking also has many interchangeable parts, such as the performance measures used, how measurement points are aggregated, and how missing values are handled. Questionable research practices could alter these elements a-posteriori to meet expectations, thereby skewing the results [27]. In the following, we discuss related work from the areas of static benchmark selection, algorithm configuration, incremental benchmark selection, and active learning. Table 1 compares the most relevant approaches, which all pursue slightly different goals. Thus, our approach is *not* a general improvement over the others but the only one fully aligned with Definition 1.

[1] https://github.com/mathefuchs/al-for-sat-solver-benchmarking
[2] https://github.com/mathefuchs/al-for-sat-solver-benchmarking-data

Table 1: Comparison of features of our benchmark-selection approach, the static benchmark-selection approach by Hoos et al. [15], the algorithm configuration system SMAC [16], and the active-learning approaches by Matricon et al. [21].

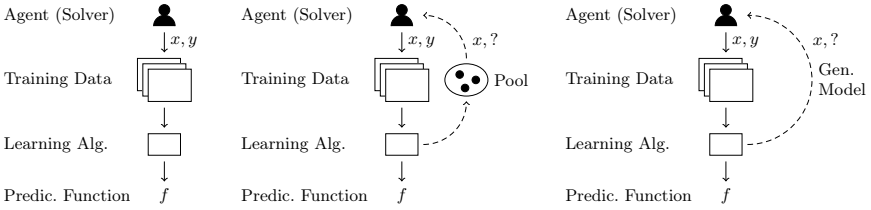
Feature	Hoos [15]	SMAC [16]	Matricon [21]	Our approach
Ranking/Scoring	✓	✗	(✓)	✓
Runtime Minimization	✗	✓	✓	✓
Incremental/Non-Random	✗	✗	✓	✓
Scalability	✓	✓	✗	✓

Static Benchmark Selection. Benchmark selection is essential for competitions, e.g., the SAT Competition. In such competitions, the organizers define the rules for composing the benchmarks. These selection strategies are primarily static, i.e., they do not depend on particular solvers to distinguish. Balint et al. provide an overview of benchmark-selection criteria in different solver competitions [1]. Froleyks et al. describe benchmark selection in recent SAT competitions [8]. Manthey and Möhle find that competition benchmarks might contain redundant instances and propose a feature-based approach to remove redundancy [20]. Mısır presents a feature-based approach to reduce benchmarks by matrix factorization and clustering [24].

Hoos et al. [15] discuss which properties are most desirable when selecting SAT benchmark instances. The selection criteria are instance variety to avoid over-fitting, adapted instance hardness (not too easy but also not too hard), and avoiding duplicate instances. To filter too similar instances, they use a distance-based approach with the SATzilla features [37,38]. The approach does, however, not optimize for benchmark *runtime* and selects instances *randomly*, apart from constraints on the instance hardness and feature distance.

Algorithm Configuration. Further related work can be found within the field of algorithm configuration [14,32], e.g., the configuration system SMAC [16]. Thereby, the goal is to tune SAT solvers for a given sub-domain of problem instances. Although this task is different from our goal, e.g., we do not need to navigate the configuration space, there are similarities to our approach as well. For example, SMAC also employs an iterative, model-based selection procedure, though for configurations rather than instances. An algorithm configurator, however, cannot be used to *rank/score* a new solver since algorithm configuration solemnly seeks to find the best-performing configuration. Also, while using a model-based selection strategy to sample configurations, instance selection is made *randomly*, i.e., without building a model over instances.

Incremental Benchmark Selection. Matricon et al. present an incremental benchmark selection approach [21]. Their *per-set efficient algorithm selection problem* (PSEAS) is similar to our *New-Solver Problem* (cf. Definition 1). Given a pair of SAT solvers, they iteratively select a subset of instances until the



(a) Passive. (b) Active Pool-based. (c) Active Synthesis-based.

Fig. 1: Types of machine learning (depiction inspired by Rubens et.al. [29]).

desired confidence level is reached to decide which of the two solvers is better. The selection of instances depends on the choice of the solvers to distinguish. They calculate a scoring metric for all unselected instances, run the experiment with the highest score, and update the confidence. Their approach ticks off most of our desired features in Table 1. However, the approach only compares solvers binarily rather than providing a *scoring*. Thus, it is unclear how similar two given solvers are or on which instances they behave similarly. Moreover, a significant shortcoming is the lacking *scalability* with the number of solvers. Comparing only pairs of solvers, evaluating a new solver requires sampling a separate benchmark for each existing solver. In contrast, our approach allows comparing a new solver against a set of existing solvers by sampling only one benchmark.

Active Learning. Prediction models in passive machine learning are trained on datasets with given instance labels (cf. Fig. 1a). In contrast, active learning (AL) starts with no or little labeled data. It repeatedly selects interesting problem instances for which to acquire labels, aiming to gradually improve the prediction model (cf. Fig. 1b). AL methods are especially beneficial if acquiring labels is computationally expensive, like obtaining solver runtimes. Without AL methods, it is not obvious which instances to label and which not. On the one hand, we want to maximize the utility an instance provides to our model, i.e., rank prediction accuracy, and on the other hand, minimize the cost, i.e., predicted runtime, associated with the instance's acquisition. Thus, we strive for an accurate prediction model without having to label every data point.

Rubens et. al. [29] survey active-learning advances. While synthesis-based AL methods [5,9,34] generate instances for labeling, pool-based methods [11,13,19] rely on a fixed set of unlabeled instances to sample from. Recent synthesis-based methods within the field of SAT solving show how to generate problem instances with desired properties [5,9]. This goal is, however, orthogonal to ours. While those approaches want to generate instances on which a solver is good or bad, we want to predict whether a solver is good or bad on an existing benchmark. Volpato and Guangyan use pool-based AL to learn an instance-specific algorithm selector [35]. Rather than benchmarking a solver's overall performance, their goal is to recommend the best solver out of a set of solvers for each SAT instance.

Algorithm 1: Incremental Benchmarking Framework

```
Input: Solvers 𝒜, Instances ℐ, Runtimes r : 𝒜 × ℐ → [0, τ], Solver â
Output: Predicted Score of â, Measured Runtimes ℛ
1 ℳ ← initModel (𝒜, ℐ, r, â)                    // cf. Section 3.1
2 ℛ ← ∅
3 while not stop (ℳ) do                          // cf. Section 3.3
4     e ← selectNextInstance (ℳ)                 // cf. Section 3.2
5     t ← runExperiment (â, e)                   // Runs â on e with timeout τ
6     ℛ ← ℛ ∪ {(e, t)}
7     updateModel (ℳ, ℛ)                         // cf. Section 3.1
8 s_â ← predictScore(ℳ)                          // cf. Section 3.1
9 return (s_â, ℛ)
```

3 Active Learning for SAT Solver Benchmarking

Algorithm 1 outlines our benchmarking framework. Given a set of solvers $\mathcal{A}$, instances $\mathcal{I}$ and runtimes r, we first initialize a prediction model $\mathcal{M}$ for the new solver $\hat{a} \notin \mathcal{A}$ (Line 1). The prediction model $\mathcal{M}$ is used to repeatedly select an instance (Line 4) for benchmarking $\hat{a}$ (Line 5). The acquired result is subsequently used to update the prediction model $\mathcal{M}$ (Line 7). When the stopping criterion is met (Line 3), we quit the benchmarking loop and predict the final score of $\hat{a}$ (Line 8). Algorithm 1 returns the predicted score of $\hat{a}$ as well as the acquired instances and runtime measurements (Line 9).

Section 3.1 describes the underlying prediction model $\mathcal{M}$ and specifies how we may derive a solver ranking from it. We discuss criteria for selecting instances in Section 3.2. Section 3.3 concludes with possible stopping conditions.

3.1 Solver Model

The model M provides a runtime-label prediction function $f : \hat{\mathcal{A}} \times \mathcal{I} \to \mathbb{R}$ for all solvers $\hat{\mathcal{A}} := \mathcal{A} \cup \{\hat{a}\}$. This prediction function powers instance selection as described in Section 3.2. During model updates (Algorithm 1, Line 7), f is trained to predict a transformed version of the acquired runtimes $\mathcal{R}$. We describe the runtime transformation in the subsequent section. The features described in Section 4.2 serve as the input to the model. Further, note that we build a new prediction model in each iteration since running experiments (Line 5) dominates the runtime of model training by magnitudes. Finally, we predict the score of the new solver $\hat{a}$ with the prediction function f (Line 8).

Runtime Transformation. For the prediction model M, we transform the real-valued runtimes into discrete runtime labels on a per-instance basis. For each instance $e \in \mathcal{I}$, we use a clustering algorithm to assign the runtimes in $\{r(a, e) \mid a \in \mathcal{A}\}$ to one of k clusters $C_1, \ldots, C_k$ such that the fastest runtimes

for the instance e are in cluster C_1 and the slowest are in cluster C_{k-1}. Timeouts τ always form a separate cluster C_k. The runtime transformation function $\gamma_k : \mathcal{A} \times \mathcal{I} \to \{1, \ldots, k\}$ is then specified as follows:

$$\gamma_k(a, e) = j \;\Leftrightarrow\; r(a, e) \in C_j$$

Given an instance $e \in \mathcal{I}$, a solver $a \in \mathcal{A}$ belongs to the $\gamma_k(a, e)$-fastest solvers on instance e. In preliminary experiments, we achieved higher accuracy for predicting such discrete runtime labels than for predicting raw runtimes. Research on portfolio solvers has also shown that discretization works well in practice [4,26].

Ranking Solvers. To determine solver ranks, we use the transformed runtimes $\gamma_k(a, e)$ in the adapted scoring function $s_k : \mathcal{A} \to [1, 2 \cdot k]$ as follows:

$$s_k(a) := \frac{1}{|\mathcal{I}|} \sum_{e \in \mathcal{I}} \gamma'_k(a, e) \qquad \gamma'_k(a, e) := \begin{cases} 2 \cdot \gamma_k(a, e) & \text{if } \gamma_k(a, e) = k \\ \gamma_k(a, e) & \text{otherwise} \end{cases} \tag{1}$$

I.e., we apply PAR-2 scoring, which is commonly used in SAT competitions [8], on the discrete labels. The scoring function s_k induces a ranking among solvers.

3.2 Instance Selection

Selecting an instance based on the model is a core functionality of our framework (cf. Algorithm 1, Line 4). In this section, we introduce two instance sampling strategies, one that minimizes uncertainty and one that maximizes information gain. Both strategies use the model's label-prediction function f and are inspired by existing work within the realms of active learning [30]. These methods require the model's predictions to include probabilities for the k discrete runtime labels. Let $f' : \hat{\mathcal{A}} \times \mathcal{I} \to [0, 1]^k$ denote this modified prediction function. In the following, the set $\tilde{\mathcal{I}} \subseteq \mathcal{I}$ denotes the instances that have already been sampled.

Uncertainty Sampling. The uncertainty sampling strategy selects the instance closest to the model's decision boundary, i.e., we select the instance $e \in \mathcal{I} \setminus \tilde{\mathcal{I}}$ that minimizes $U(e)$, which is specified as follows:

$$\mathrm{U}(e) := \left| \frac{1}{k} - \max_{n \in \{1, \ldots, k\}} f'(\hat{a}, e)_n \right|$$

Information-Gain Sampling. The information-gain sampling strategy selects the instance with the highest expected entropy reduction regarding the runtime labels of the instance. To be more specific, we select the instance $e \in \mathcal{I} \setminus \tilde{\mathcal{I}}$ that maximizes $IG(e)$, which is specified as follows:

$$\mathrm{IG}(e) := \mathrm{H}(e) - \sum_{n=1}^{k} f'(\hat{a}, e)_n \, \hat{\mathrm{H}}_n(e)$$

Here, $\mathrm{H}(e)$ denotes the entropy of the runtime labels $\gamma(a, e)$ over all $a \in \mathcal{A}$ and $\mathrm{H}(e, n)$ denotes the entropy of these labels plus n as the runtime label for $\hat{a}$. The term $\hat{\mathrm{H}}_n(e)$ is computed for every possible runtime label $n \in \{1, \ldots, k\}$. By maximizing information gain, we select instances that identify solvers with similar behavior.

3.3 Stopping Criteria

In this section, we present the two dynamic stopping criteria in our experiments, the Wilcoxon and the ranking stopping criterion (cf. Algorithm 1, Line 3).

Wilcoxon Stopping Criterion. The Wilcoxon stopping criterion stops the active-learning process when we are confident enough that the predicted runtime labels of the new solver are sufficiently different from existing solvers. This criterion is loosely inspired by Matricon et. al. [21]. We use the average p-value $W_{\hat{a}}$ of a Wilcoxon signed-rank test $\mathrm{w}(S, P)$ of the two runtime label distributions $S = \{\gamma(a, e) \mid e \in \mathcal{I}\}$ for an existing solver a and $P = \{f(\hat{a}, e) \mid e \in \mathcal{I}\}$ for the new solver $\hat{a}$:

$$W_{\hat{a}} := \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} \mathrm{w}(S, P)$$

To improve the stability of this criterion, we use an exponential moving average to smooth out outliers and stop as soon as $W^{(i)}_{\mathrm{exp}}$ drops below a fixed threshold:

$$W^{(0)}_{\mathrm{exp}} := 1$$
$$W^{(i)}_{\mathrm{exp}} := \beta W_{\hat{a}} + (1 - \beta)\, W^{(i-1)}_{\mathrm{exp}}$$

Ranking Stopping Criterion. The ranking stopping criterion is less sophisticated in comparison. It stops the active-learning process if the ranking induced by the model's predictions (Equation 1) remained unchanged within the last l iterations. However, the concrete values of the predicted score $s_{\hat{a}}$ might still change. We are solemnly interested in the induced ranking in this case.

4 Experimental Design

Given all the previously presented instantiations for Algorithm 1, this section outlines our experimental design, including our evaluation framework, used data sets, hyper-parameter choices, and implementation details.

4.1 Evaluation Framework

As stated in the Introduction, this work addresses the *New-Solver Problem* (cf. Definition 1). As described in Section 3.1, a prediction model $\mathcal{M}$ provides us with an estimated scoring $s_{\hat{a}}$ for the new solver $\hat{a}$.

Algorithm 2: Evaluation Framework

```
Input: Solvers 𝒜, Instances ℐ, Runtimes r : 𝒜 × ℐ → [0, τ]
Output: Average Ranking Accuracy Ō_acc, Average Fraction of Runtime Ō_rt
1  O ← ∅
2  for â ∈ 𝒜 do
3      𝒜' ← 𝒜 \ {â}
4      (s_â, ℛ) ← runALAlgorithm(𝒜', ℐ, r, â)        // Refer to Algorithm 1
       // Determine Ranking Accuracy
5      O_acc ← 0
6      for a ∈ 𝒜 do
7          if (s_k(a) − s_â) · (par_2(a) − par_2(â)) > 0 then
8              O_acc ← O_acc + 1/|𝒜|
       // Determine Runtime Fraction
9      r ← Σ_{e∈ℐ} r(â, e)
10     O_rt ← 0
11     for e ∈ ℐ do
12         if ∃t, (e, t) ∈ ℛ then
13             O_rt ← O_rt + t/r
14     O ← O ∪ {(O_acc, O_rt)}
15 (Ō_acc, Ō_rt) ← average(O)
16 return (Ō_acc, Ō_rt)
```

To evaluate a concrete instantiation of Algorithm 1, i.e., a concrete choice for all the sub-routines, we perform cross-validation on our set of solvers. Algorithm 2 shows this. That means each solver plays the role of the new solver $\hat{a}$ once (Line 2). Note that the *new* solver in each iteration is excluded from the set of solvers $\mathcal{A}$ to avoid data leakage (Line 3). After running our active-learning framework for solver $\hat{a}$ (Line 4), we compute the value of both our optimization goals, i.e., ranking accuracy and runtime. We define the *ranking accuracy* $O_{\text{acc}} \in [0,1]$ (higher is better) by the fraction of pairs $(\hat{a}, a)$ for all $a \in \mathcal{A}$ that are decided correctly regarding the ground-truth scoring par_2 (Lines 5-8). The *fraction of runtime* that the algorithm needs to arrive at its conclusion is denoted by $O_{\text{rt}} \in [0,1]$ (lower is better). This metric puts the runtime summed over the sampled instances in relation to the runtime summed over all instances in the dataset (Lines 9-13). Finally, we compute averages of the output metrics in Line 15 after we have collected all cross-validation results in Line 14. Overall, we want to find an approach that maximizes

$$O_\delta := \delta O_{\text{acc}} + (1-\delta)\,(1 - O_{\text{rt}}) \quad , \tag{2}$$

whereby $\delta \in [0,1]$ allows for linear weighting between the two optimization goals O_{acc} and O_{rt}. Plotting the approaches that maximize O_δ for all $\delta \in [0,1]$ on

an O_{rt}-O_{acc}-diagram provides us with a Pareto front of the best approaches for different optimization-goal weightings.

4.2 Data

In our experiments, we work with the dataset of the SAT Competition 2022 Anniversary Track [2]. The dataset consists of 5355 instances with respective runtime data of 28 sequential SAT solvers. We also use a database of 56 instance features[3] from the Global Benchmark Database (GBD) by Iser et al. [17]. They comprise instance size features and node distribution statistics for several graph representations of SAT instances, among others, and are primarily inspired by the SATzilla 2012 features described in [38]. All features are numeric and free of missing values. We drop 10 out of 56 features because of zero variance. Overall, prediction models have access to 46 instance features and 27 runtime features, i.e., excluding the current new solver $\hat{a}$.

Additionally, we retrieve instance-family information[4] to evaluate the composition of our sampled benchmarks. Instance families comprise instances from the same application domain, e.g., planning, cryptography, etc., and are a valuable tool for analyzing solver performance.

For hyper-parameter tuning, we randomly sample 10 % of the complete set of 5355 instances with stratification regarding the instances' family. All instance families that are too *small*, i.e., 10 % of them corresponds to less than one instance, are put into one meta-family for stratification. This *tuning dataset* allows for a more extensive exploration of the hyper-parameter space.

4.3 Hyper-parameters

Given Algorithm 1, there are several possible instantiations for the three subroutines, i.e., *ranking*, *selection*, and *stopping*. Also, there are different choices for the runtime-label prediction model and runtime discretization. We describe these experimental configurations in the following.

Ranking. Regarding *ranking* (cf. Section 3.1), we experiment with the following approaches and hyper-parameter values:

- Observed PAR-2 ranking of already sampled instances
- Predicted runtime-label ranking
  - History size: Consider the latest 1, 10, 20, 30, or 40 predictions within a voting approach for stability. The latest x predictions for each instance vote on the instance's winning label.
  - Fallback threshold: If the difference of scores between the new solver $\hat{a}$ and another solver drops below 0.01 , 0.05 , or 0.1 , use the partially observed PAR-2 ranking as a tie-breaker.

[3] https://benchmark-database.de/getdatabase/base_db
[4] https://benchmark-database.de/getdatabase/meta_db

Selection. For *selection* (cf. Section 3.2), we experiment with the following methods and hyper-parameter values. Since the potential runtime of experiments is by magnitudes larger than the model's update time, we only consider incrementing our benchmark by one instance at a time rather than using batches, which is also proposed in current active-learning advances [31,34]. A drawback of this is the lack of parallel execution of runtime experiments.

- Random sampling
- Uncertainty sampling
  - Fallback threshold: Use random sampling for the first 0 %, 5 %, 10 %, 15 %, or 20 % of instances to explore the instance space.
  - Runtime scaling: Whether to normalize uncertainty scores per instance by the average runtime of solvers on it or use the absolute values.
- Information-gain sampling
  - Fallback threshold: Use random sampling for the first 0 %, 5 %, 10 %, 15 %, or 20 % of instances to explore the instance space.
  - Runtime scaling: Whether to normalize information-gain scores per instance by the average runtime of solvers on it or use the absolute values.

Stopping. For *stopping* decisions (cf. Section 3.3), we experiment with the following criteria and hyper-parameter values:

- Subset-size stopping criterion, using 10 % or 20 % of instances
- Ranking stopping criterion
  - Minimum amount: Sample at least 2 %, 8 %, 10 %, or 12 % of instances before applying the criterion.
  - Convergence duration: Stop if the predicted ranking stays the same for a number of sampled instances equal to 1 % or 2 % of all instances.
- Wilcoxon stopping criterion
  - Minimum amount: Sample at least 2 %, 8 %, 10 %, or 12 % of instances before applying the criterion.
  - Average of p-values to drop below: 5 %.
  - Exponential-moving average: Incorporate previous significance values by using an EMA with $\beta = 0.1$ or $\beta = 0.7$.

Prediction model. Our experiments only use one model configuration for runtime-label prediction since an exhaustive grid search would be infeasible. In preliminary experiments, we compared various model types from *scikit-learn* [28]. In particular, we conducted nested cross-validation, including hyper-parameter tuning, and used Matthews Correlation Coefficient [12,22] to assess the performance for predicting runtime labels. Our final choice is a stacking ensemble [36] of two prediction models, a quadratic-discriminant analysis [33] and a random forest [3]. Both these models can learn non-linear relationships between the instance features and the runtime labels. Stacking means that another prediction model, in our case a simple decision tree, decides which of the two ensemble members makes the prediction on which instance.

Runtime discretization. To define prediction targets, i.e., discrete runtime labels, we use hierarchical clustering with $k = 3$ and a log-single-link criterion, which produced the most *useful* labels in preliminary experiments. We denote this adapted solver scoring function with s_3. In our chosen hierarchical procedure, each non-timeout runtime starts in a separate interval. We then gradually merge intervals whose single-link logarithmic distance is the smallest until the desired number of partitions is reached. Other clustering approaches that we tried include hierarchical clustering with mean-, median-, and complete-link criterion, as well as k-means and spectral clustering.

To obtain *useful* labels, we need to ensure that discretized labels still discriminate solvers and align with the actual PAR-2 ranking. We analyzed the ranking induced by s_3 in preliminary experiments with the SAT Competition 2022 Anniversary Track [2]. According to a Wilcoxon-signed-rank test with $\alpha = 0.05$, 87.83 % of solver pairs have significantly different scores after discretization, only a slight drop compared to 89.95 % before discretization. Further, our ranking approach correctly decides for almost all (about 97.45 %; $\sigma = 3.68$ %) solver pairs which solver is faster. In particular, the Spearman correlation of s_3 and PAR-2 ranking is about 0.988, which is very close to the optimal value of 1 [6]. All these results show that discretized runtimes are suitable for our framework.

4.4 Implementation Details

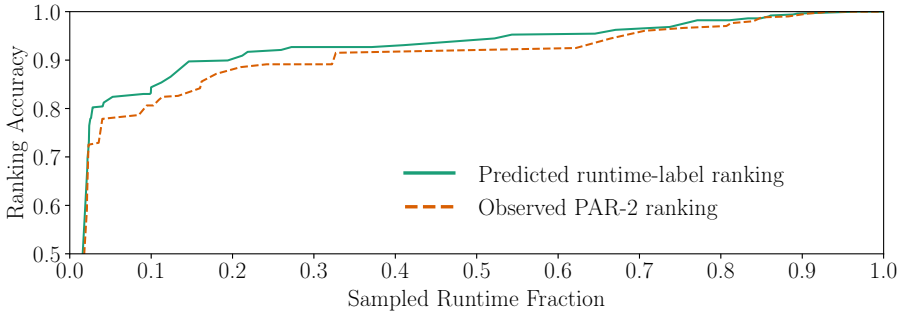
For reproducibility, our source code and data are available on GitHub (cf. footnotes in Section 1). Our code is implemented in Python using *scikit-learn* [28] for making predictions and *gbd-tools* [17] for SAT-instance retrieval.

5 Evaluation

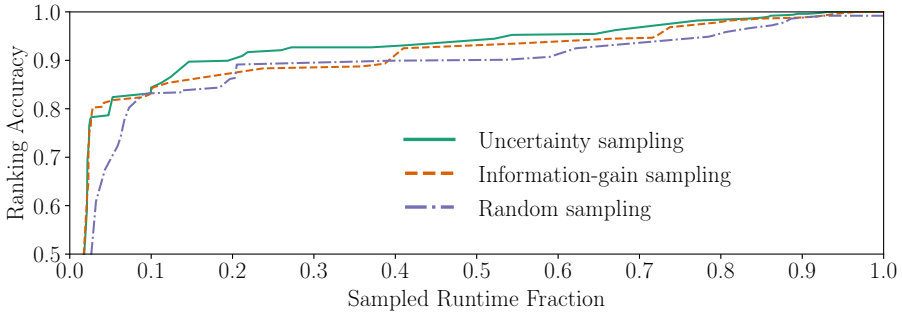
In this section, we evaluate our active-learning framework. First, we analyze and tune the different sub-routines of our framework on the tuning dataset. Next, we evaluate the best configurations with the full dataset. Finally, we analyze the importance of different instance families to our framework.

5.1 Hyper-Parameter Analysis

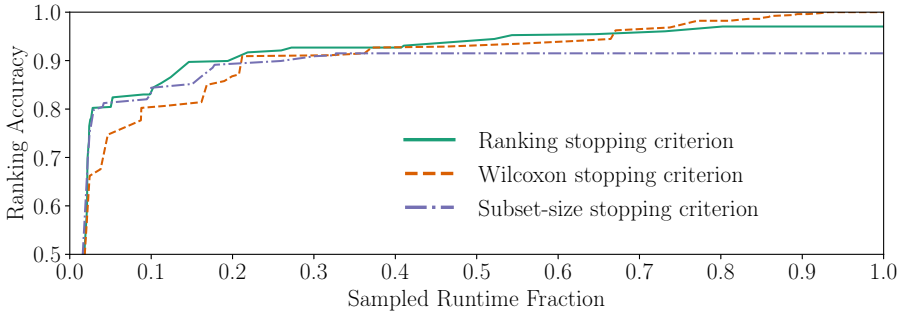
Our experiments follow the evaluation framework introduced in Section 4.1. Fig. 2 shows the performance of the approaches from Section 4.3 on O_{rt}-O_{acc}-diagrams for the hyper-parameter-tuning dataset. Evaluating a particular configuration with Algorithm 2 returns a point $(O_{\text{rt}}, O_{\text{acc}})$. We do not show intermediate results of the active-learning procedure but only the final results after stopping. The plotted lines represent the best-performing configurations per ranking approach (Fig. 2a), selection approach (Fig. 2b), and stopping criterion (Fig. 2c). In particular, we show the Pareto front, i.e., of all configurations that share a particular value of the plotted hyper-parameter, we take the maximum ranking accuracy over all remaining hyper-parameters *not* displayed in the corresponding plot.



(a) Ranking approaches

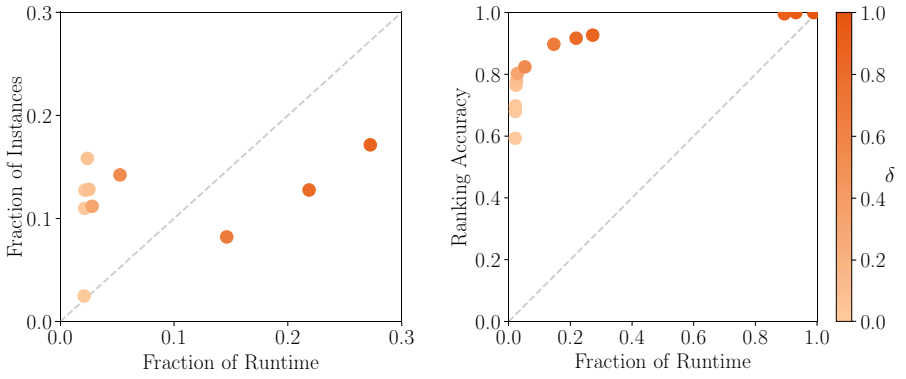


(b) Selection approaches



(c) Stopping criteria

Fig. 2: O_{rt}-O_{acc}-diagrams comparing different hyper-parameter instantiations of our active-learning framework on the hyper-parameter-tuning dataset. The x-axis shows the ratio of total solver runtime on the sampled instances relative to all instances. The y-axis shows the ranking accuracy (cf. Section 4.1). Each line entails the front of Pareto-optimal configurations for the respective hyper-parameter instantiation.



(a) Runtime vs. Instances (b) Runtime vs. Accuracy

Fig. 3: Scatter plot comparing different instantiations of trade-off parameter δ for our active-learning framework on the hyper-parameter-tuning dataset. The x-axis shows the fraction of runtime O_{rt} of the sample, while the y-axes show the fraction of instances sampled and ranking accuracy, respectively. The color indicates the weighting between different optimization goals $\delta \in [0, 1]$. The larger δ, the more we favor accuracy over runtime.

Regarding ranking approaches (Fig. 2a), using the predicted s_3-induced runtime-label ranking consistently outperforms the partially observed PAR-2 ranking for each possible value of the trade-off parameter δ. This outcome is expected since selection decisions are not random. For example, we might sample more instances of one family if it benefits discrimination of solvers. While the partially observed PAR-2 score is skewed, the prediction model can account for this.

Regarding the selection approaches (Fig. 2b), uncertainty sampling performs best in most cases. However, information-gain sampling is beneficial if runtime is strongly favored (small δ; runtime fraction less than 5 %). This result aligns with our expectations: Information-gain sampling selects instances that maximize the expected reduction in entropy. This means we sample instances revealing similarities between solvers rather than differences, which helps to build a confident model quickly. However, the method cannot select helpful instances for distinguishing solvers later. Random sampling performs reasonably well but is outperformed by uncertainty sampling in all cases, showing the benefit of actively selecting instances based on a prediction model.

Regarding the stopping criteria (Fig. 2c), the ranking stopping criterion performs most consistently well. If accuracy is strongly favored (very high δ), the Wilcoxon stopping criterion performs better. The subset-size stopping criterion performs reasonably well but does not improve beyond a certain accuracy because of sampling a fixed subset of instances.

Fig. 3a shows an interesting consequence of weighting our optimization goals: If we, on the one hand, desire to get a *rough* estimate of a solver's performance

Table 2: Performance comparison (on the full dataset) of the best-performing active-learning approaches (*AL*), random sampling of the same runtime fraction with 1000 repetitions (*Random*), and statically selecting the instances most frequently sampled by active-learning approaches (*Most Freq.*)

(a) Best-performing AL approach for $\delta \in [0.2, 0.7]$

	AL	Random	Most Freq.
Sampled Runtime Fraction (%)	5.41	5.43	5.44
Sampled Instance Fraction (%)	26.53	5.43	27.75
Ranking Accuracy (%)	90.48	88.54	81.08

(b) Best-performing AL approach for $\delta \in (0.7, 0.8]$

	AL	Random	Most Freq.
Sampled Runtime Fraction (%)	10.35	10.37	10.37
Sampled Instance Fraction (%)	5.24	10.37	36.96
Ranking Accuracy (%)	92.33	91.61	84.52

fast (low δ), approaches favor selecting many *easy* instances. In particular, the fraction of sampled instances is larger than the fraction of runtime. By having many observations, it is easier to build a model. If we, on the other hand, desire to get a *good* estimate of a solver's performance in a moderate amount of time (high δ), approaches favor selecting few, *difficult* instances. In particular, the fraction of instances is smaller than the fraction of runtime.

Furthermore, Fig. 3b reveals which values make the most sense for δ. The range $\delta \in [0.2, 0.8]$, thereby, corresponds to the points with a runtime fraction between 0.03 and 0.22 We consider this region to be most promising, analogous to the *elbow* method in cluster analysis [18].

5.2 Full-Dataset Evaluation

Having selected the most promising hyper-parameters, we run our active-learning experiments on the complete Anniversary Track dataset (5355 instances). The aforementioned range $\delta \in [0.2, 0.8]$ only results in two distinct configurations. The best-performing approach for $\delta \in [0.2, 0.7]$ uses the predicted runtime-label ranking, information-gain sampling, and ranking stopping criterion. It can predict a new solver's PAR-2 ranking with 90.48 % accuracy (O_{acc}) in only 5.41 % of the full evaluation time (O_{rt}). The best-performing approach for $\delta \in (0.7, 0.8]$ uses the predicted runtime-label ranking, uncertainty sampling, and ranking stopping criterion. It can predict a new solver's PAR-2 ranking with 92.33 % accuracy (O_{acc}) in only 10.35 % of the full evaluation time (O_{rt}).

Table 2 shows how both active-learning approaches (column *AL*) compare against two static baselines: *Random* samples instances until it reaches roughly the same fraction of runtime as the AL benchmark sets. We repeat sampling 1000 times and report average results. *Most Freq.* uses a static benchmark set

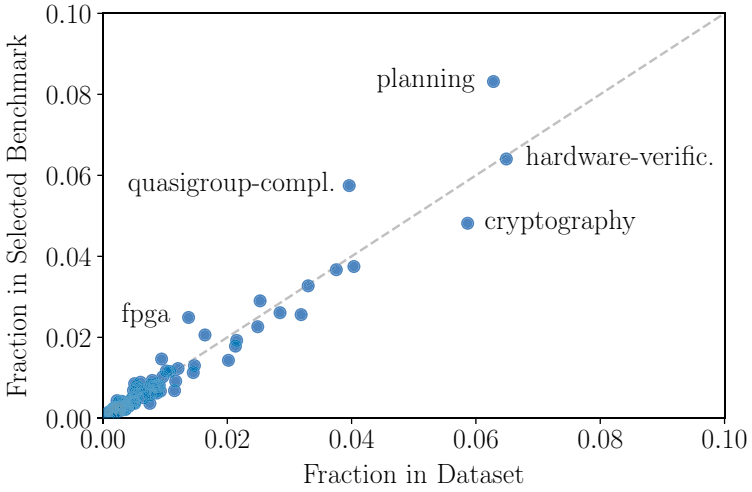


Fig. 4: Scatter plot showing the *importance* of different instance families to our framework on the full dataset. The x-axis shows the frequency of instance families in the dataset. The y-axis shows the average frequency of instance families in the samples selected by active learning. The dashed line represents families that occur with the same frequency in the dataset and samples.

consisting of those instances most frequently sampled by our active learning approach. In particular, we consider the average sampling frequency over all solvers and Pareto-optimal active-learning approaches.

Both our AL approaches perform better than random sampling. However, the performance differences are not significant regarding a Wilcoxon signed-rank test with $\alpha = 0.05$ and also depend on the fraction of sampled runtime (cf. Fig. 2b). A clear advantage of our approach is, though, that it indicates when to stop adding further instances, depending on the trade-off parameter δ. While the active-learning results are less strong on the full dataset than on the smaller tuning dataset, they still show the benefit of making benchmark selection dependent on the solvers to distinguish.

A static benchmark using the most frequently AL-sampled instances performs poorly, though, compared to active learning and random sampling. This outcome is somewhat expected since the static benchmark does not reflect the right balance of instance families: Families whose instances are uniform-randomly selected by AL, e.g., for different solvers, appear less often in this benchmark than families where some instances are sampled more often than others.

5.3 Instance-Family Importance

Selection decisions of our approach also reveal the importance of different instance families to our framework. Fig. 4 shows the occurrence of instance fami-

lies within the dataset and the benchmarks created by active learning. We use the best-performing configurations for all $\delta \in [0, 1]$ and examine the selection decisions by the active-learning approach on the SAT Competition 2022 Anniversary Track dataset [2]. While most families appear with the same fraction in the dataset and the sampled benchmarks, a few outliers need further discussion. Problem instances of the families *fpga*, *quasigroup-completion*, and *planning* are especially helpful to our framework in distinguishing solvers. Instances of these families are selected over-proportionally in comparison to the full dataset. In contrast, instances of the largest family, i.e., *hardware-verification*, roughly appear with the same fraction in the dataset and the sampled benchmarks. Finally, instances of the family *cryptography* are less important in distinguishing solvers than their vast weight in the dataset suggests. A possible explanation is that these instances are very similar, such that a small fraction of them is sufficient to estimate a solver's performance on all of them.

6 Conclusions and Future Work

In this work, we have addressed the *New-Solver Problem*: Given a new solver, we want to find its ranking amidst competitors. Our approach provides accurate ranking predictions while needing significantly less runtime than a complete evaluation on a given benchmark set. On data from the SAT Competition 2022 Anniversary Track, we can determine a new solver's PAR-2 ranking with about 92 % accuracy while only needing 10 % of the full-evaluation time. We have evaluated several ranking algorithms, instance-selection approaches, and stopping criteria within our sequential active-learning framework. We also took a brief look at which instance families are the most prevalent in selection decisions.

Future work may compare further sub-routines for ranking, instance selection, and stopping. Additionally, one can apply our evaluation framework to arbitrary computation-intensive problems, e.g., other $\mathcal{NP}$-complete problems than SAT, as all discussed active-learning methods are problem-agnostic. Such problems share most of the relevant properties of SAT solving, i.e., there are established instance features, a complete benchmark is expensive, and traditional benchmark selection requires expert knowledge.

From the technical perspective, one could formulate runtime discretization as an optimization problem rather than addressing it empirically. Further, a major shortcoming of our current approach is the lack of parallelization, selecting instances one at a time. Benchmarking on a computing cluster with n cores benefits from having batches of n instances. However, bigger batch sizes n impede *active learning*. Also, it is unclear how to synchronize instance selection and updates of the prediction model without wasting too much runtime.

Acknowledgments. This work was supported by the Ministry of Science, Research and the Arts Baden-Württemberg, project *Algorithm Engineering for the Scalability Challenge (AESC)*.

References

1. Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Overview and analysis of the SAT Challenge 2012 solver competition. Artif. Intell. **223**, 120–155 (2015). https://doi.org/10.1016/j.artint.2015.01.002
2. Balyo, T., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions. Department of Computer Science, University of Helsinki (2022), http://hdl.handle.net/10138/347211
3. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324
4. Collautti, M., Malitsky, Y., Mehta, D., O'Sullivan, B.: SNNAP: solver-based nearest neighbor for algorithm portfolios. In: Proc. ECML PKDD. pp. 435–450 (2013). https://doi.org/10.1007/978-3-642-40994-3_28
5. Dang, N., Akgün, Ö., Espasa, J., Miguel, I., Nightingale, P.: A framework for generating informative benchmark instances. In: Proc. CP. pp. 18:1–18:18 (2022). https://doi.org/10.4230/LIPIcs.CP.2022.18
6. De Winter, J.C.F., Gosling, S.D., Potter, J.: Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data. Psychol. Methods **21**(3), 273–290 (2016). https://doi.org/10.1037/met0000079
7. Dehghani, M., Tay, Y., Gritsenko, A.A., Zhao, Z., Houlsby, N., Diaz, F., Metzler, D., Vinyals, O.: The benchmark lottery. arXiv:2107.07002 [cs.LG] (2021), https://arxiv.org/abs/2107.07002
8. Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M.: SAT Competition 2020. Artif. Intell. **301** (2021). https://doi.org/10.1016/j.artint.2021.103572
9. Garzón, I., Mesejo, P., Giráldez-Cru, J.: On the performance of deep generative models of realistic SAT instances. In: Proc. SAT. pp. 3:1–3:19 (2022). https://doi.org/10.4230/LIPIcs.SAT.2022.3
10. Gelder, A.V.: Careful ranking of multiple solvers with timeouts and ties. In: Proc. SAT. pp. 317–328 (2011). https://doi.org/10.1007/978-3-642-21581-0_25
11. Golbandi, N., Koren, Y., Lempel, R.: Adaptive bootstrapping of recommender systems using decision trees. In: Proc. WSDM. pp. 595–604 (2011). https://doi.org/10.1145/1935826.1935910
12. Gorodkin, J.: Comparing two k-category assignments by a k-category correlation coefficient. Comput. Biol. Chem. **28**(5–6), 367–374 (2004). https://doi.org/10.1016/j.compbiolchem.2004.09.006
13. Harpale, A., Yang, Y.: Personalized active learning for collaborative filtering. In: Proc. SIGIR. pp. 91–98 (2008). https://doi.org/10.1145/1390334.1390352
14. Hoos, H.H., Hutter, F., Leyton-Brown, K.: Automated configuration and selection of SAT solvers. In: Handbook of Satisfiability, chap. 12, pp. 481–507. IOS Press, 2 edn. (2021). https://doi.org/10.3233/FAIA200995
15. Hoos, H.H., Kaufmann, B., Schaub, T., Schneider, M.: Robust benchmark set selection for boolean constraint solvers. In: Proc. LION. pp. 138–152 (2013). https://doi.org/10.1007/978-3-642-44973-4_16
16. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proc. LION. pp. 507–523 (2011). https://doi.org/10.1007/978-3-642-25566-3_40
17. Iser, M., Sinz, C.: A problem meta-data library for research in SAT. In: Proc. PoS. pp. 144–152 (2018). https://doi.org/10.29007/gdbb

18. Kodinariya, T.M., Makwana, P.R.: Review on determining number of cluster in k-means clustering. Int. J. Adv. Res. Comput. Sci. Manage. Stud. **1**(6), 90–95 (2013), http://www.ijarcsms.com/docs/paper/volume1/issue6/V1I6-0015.pdf
19. Koren, Y., Bell, R.M., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **42**(8), 30–37 (2009). https://doi.org/10.1109/MC.2009.263
20. Manthey, N., Möhle, S.: Better evaluations by analyzing benchmark structure. In: Proc. PoS (2016), http://www.pragmaticsofsat.org/2016/reg/POS-16_paper_4.pdf
21. Matricon, T., Anastacio, M., Fijalkow, N., Simon, L., Hoos, H.H.: Statistical comparison of algorithm performance through instance selection. In: Proc. CP. pp. 43:1–43:21 (2021). https://doi.org/10.4230/LIPIcs.CP.2021.43
22. Matthews, B.W.: Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Biochim. Biophys. Acta - Protein Struct. **405**(2), 442–451 (1975). https://doi.org/10.1016/0005-2795(75)90109-9
23. Mısır, M.: Data sampling through collaborative filtering for algorithm selection. In: Proc. IEEE CEC. pp. 2494–2501 (2017). https://doi.org/10.1109/CEC.2017.7969608
24. Mısır, M.: Benchmark set reduction for cheap empirical algorithmic studies. In: Proc. IEEE CEC. pp. 871–877 (2021). https://doi.org/10.1109/CEC45853.2021.9505012
25. Mısır, M., Sebag, M.: ALORS: An algorithm recommender system. Artif. Intell. **244**, 291–314 (2017). https://doi.org/10.1016/j.artint.2016.12.001
26. Ngoko, Y., Cérin, C., Trystram, D.: Solving SAT in a distributed cloud: A portfolio approach. Int. J. Appl. Math. Comput. Sci. **29**(2), 261–274 (2019). https://doi.org/10.2478/amcs-2019-0019
27. Nießl, C., Herrmann, M., Wiedemann, C., Casalicchio, G., Boulesteix, A.: Over-optimism in benchmark studies and the multiplicity of design and analysis options when interpreting their results. WIREs Data Min. Knowl. Discov. **12**(2) (2022). https://doi.org/10.1002/widm.1441
28. 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., Édouard Duchesnay: Scikit-learn: Machine learning in Python. J. Mach. Learn. Res. **12**(85), 2825–2830 (2011), http://jmlr.org/papers/v12/pedregosa11a.html
29. Rubens, N., Elahi, M., Sugiyama, M., Kaplan, D.: Active learning in recommender systems. In: Recommender Systems Handbook, chap. 24, pp. 809–846. Springer, 2 edn. (2015). https://doi.org/10.1007/978-1-4899-7637-6_24
30. Settles, B.: Active learning literature survey. Tech. rep., University of Wisconsin-Madison, Department of Computer Sciences (2009), http://digital.library.wisc.edu/1793/60660
31. Sinha, S., Ebrahimi, S., Darrell, T.: Variational adversarial active learning. In: Proc. ICCV. pp. 5971–5980 (2019). https://doi.org/10.1109/ICCV.2019.00607
32. Stützle, T., López-Ibáñez, M., Pérez-Cáceres, L.: Automated algorithm configuration and design. In: Proc. GECCO. pp. 997–1019 (2022). https://doi.org/10.1145/3520304.3533663
33. Tharwat, A.: Linear vs. quadratic discriminant analysis classifier: a tutorial. Int. J. Appl. Pattern Recognit. **3**(2), 145–180 (2016). https://doi.org/10.1504/IJAPR.2016.079050

34. Tran, T., Do, T., Reid, I.D., Carneiro, G.: Bayesian generative active deep learning. In: Proc. ICML. pp. 6295–6304 (2019), http://proceedings.mlr.press/v97/tran19a.html
35. Volpato, R., Song, G.: Active learning to optimise time-expensive algorithm selection. arXiv:1909.03261 [cs.LG] (2019), https://arxiv.org/abs/1909.03261
36. Wolpert, D.H.: Stacked generalization. Neural Networks **5**(2), 241–259 (1992). https://doi.org/10.1016/S0893-6080(05)80023-1
37. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008). https://doi.org/10.1613/jair.2490
38. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Features for SAT. Tech. rep., University of British Columbia (2012), https://www.cs.ubc.ca/labs/beta/Projects/SATzilla/Report_SAT_features.pdf


ParaQooba: A Fast and Flexible Framework for Parallel and Distributed QBF Solving*

Maximilian Heisinger[1](✉), Martina Seidl[1], and Armin Biere[2]

[1] JKU Linz, Linz, Austria, {maximilian.heisinger,martina.seidl}@jku.at
[2] ALU Freiburg, Freiburg, Germany, biere@informatik.uni-freiburg.de

Abstract. Over the last years, innovative parallel and distributed SAT solving techniques were presented that could impressively exploit the power of modern hardware and cloud systems. Two approaches were particularly successful: (1) search-space splitting in a Divide-and-Conquer (D&C) manner and (2) portfolio-based solving. The latter executes different solvers or configurations of solvers in parallel. For quantified Boolean formulas (QBFs), the extension of propositional logic with quantifiers, there is surprisingly little recent work in this direction compared to SAT. In this paper, we present ParaQooba, a novel framework for parallel and distributed QBF solving which combines D&C parallelization and distribution with portfolio-based solving. Our framework is designed in such a way that it can be easily extended and arbitrary sequential QBF solvers can be integrated out of the box, without any programming effort. We show how ParaQooba orchestrates the collaboration of different solvers for joint problem solving by performing an extensive evaluation on benchmarks from QBFEval'22, the most recent QBF competition.

1 Introduction

Quantified Boolean formulas (QBFs) extend propositional logic by quantifiers over the Boolean variables [2]. As a consequence, the decision problem of QBF (QSAT) is PSPACE complete, which is potentially harder than the NP-complete decision problem of propositional logic (SAT). Hence, the quantifiers allow for an efficient encoding of many reasoning problems from formal verification, synthesis, and planning [26] that most likely do not have a compact formulation in propositional logic. Over the last decade, considerable progress has been made in sequential QBF solving [22,21]. In contrast to SAT, where conflict-driven clause learning (CDCL) [19] is the predominant solving paradigm, in QBF solving different approaches of orthogonal strength have been presented. Besides QCDCL, the QBF variant of CDCL, which is implemented for example in the solver DepQBF [17], clausal abstraction as implemented in the solver Caqe [23] and abstraction-refinement based expansion as implemented in the solver RaReQs [13] are particularly successful [22,21]. All of these QBF solving approaches considerably benefit from preprocessing, i.e., an extra step before

* Supported by the LIT AI Lab funded by the State of Upper Austria.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 426–447, 2023.
https://doi.org/10.1007/978-3-031-30823-9_22

the actual solving in which certain redundancies of a formula are eliminated in a satisfiability-preserving way with the aim to make it easier for the solver [10].

Despite the vivid development in sequential QBF solving, only few approaches have been presented for parallel and distributed QBF solving [18]. The most recent parallel QBF solvers are HordeQBF [1] which integrates sequential QCDCL-based solvers to obtain a parallel QBF solver and, more recently, a basic implementation of a QBF module based on the parallel SAT solver ParaCooba [6] with DepQBF as its only backend solver. To the best of our knowledge, besides these two approaches no other parallel QBF solver has recently been presented. The situation in SAT is different: several very powerful parallel and distributed SAT solvers like Mallob [24], Painless [5], and the afore mentioned solver ParaCooba [7] have been released. They show the potential of parallel and distributed approaches impressively by solving hard SAT instances, for example from multiplier verification [15].

In this paper, we present ParaQooba, a novel framework for parallel and distributed QBF solving that integrates search-space splitting based on the Divide-and-Conquer paradigm with portfolio solving. Our framework is built on top of the ParaCooba SAT solving framework and extends its basic non-portfolio QBF solving module. ParaQooba reuses most of ParaCooba's modules providing management and distribution of solver tasks. In addition, we implemented a very generic interface that allows the easy integration of any QBF solver binary into our framework.

Our main contributions are as follows:

- we present a new flexible framework for parallel and distributed QBF solving that combines D&C search-space splitting with portfolio solving;
- we show how different QBF solvers that are based on different solving approaches can be integrated seamlessly into our framework;
- we provide our framework as open-source project;
- we perform an extensive evaluation that demonstrates the power of our approach on various kinds of benchmarks.

ParaQooba is integrated into ParaCooba's and available on GitHub:

https://github.com/maximaximal/paracooba

This paper is structured as follows: First we introduce some preliminaries required for the rest of the paper in the following section. We continue with related work in section 3. After that, section 4 summarizes concepts of the ParaCooba solver framework used in our work. Then we introduce how we apply Divide-and-Conquer to solving QBF in section 5. Having introduced the background, we present our portfolio ParaQooba module in detail in section 6 and provide an extensive evaluation in section 7. Finally, we summarize our findings and conclude in section 8.

2 Preliminaries

We consider QBFs $\mathcal{Q}.\varphi$ in *prenex conjunctive normal form* (PCNF) where the *prefix* $\mathcal{Q}$ is of the form $Q_1 x_1, \ldots, Q_n x_n$ with $Q \in \{\forall, \exists\}$. The *matrix* φ is a propositional formula over the variables $x_1, \ldots, x_n$ in conjunctive normal form (CNF). A formula in CNF is a conjunction ($\wedge$) of clauses. A *clause* is a disjunction ($\vee$) of literals. A literal is a variable x, a negated variable $\neg x$ or a (possibly negated) truth constant $\top$ (true) or $\bot$ (false). For a literal l, the expression $\bar{l}$ denotes x if $l = \neg x$ and it denotes $\neg x$ otherwise. We sometimes write a clause as a set of literals and a CNF formula as set of clauses. Further, it is often convenient to partition the quantifier prefix into *quantifier blocks*, i.e., maximal sets of consecutive sets of variables with the same quantifier type. For example, for the QBF $\forall x_1 \forall x_2 \exists y_1 \exists y_2 . \varphi$ we also write $\forall X \exists Y . \varphi$ with $X = \{x_1, x_2\}$ and $Y = \{y_1, y_2\}$. With upper case letters $X, Y, \ldots$ (possibly subscripted), we usually denote sets of variables, while with lower case letters $x, y, \ldots$ (also possibly subscripted), we denote variables. If φ is CNF formula, then $\varphi_{x \leftarrow t}$ is the CNF formula obtained from φ by replacing all occurrences of variable x by truth constant $t \in \{\top, \bot\}$. Depending on the value of t, variable x is either set to true (if t is $\top$) or to false (if t is $\bot$). We define the semantics of QBFs as follows:

- a QBF $\forall X \mathcal{Q}.\varphi$ is true iff both QBFs $\forall X' \mathcal{Q}.\varphi_{x \leftarrow \bot}$ and $\forall X' \mathcal{Q}.\varphi_{x \leftarrow \top}$ are true where $x \in X$ and $X' = X \setminus \{x\}$;
- a QBF $\exists Y \mathcal{Q}.\varphi$ is true iff at least one of $\exists Y' \mathcal{Q}.\varphi_{y \leftarrow \bot}$ and $\exists Y' \mathcal{Q}.\varphi_{y \leftarrow \top}$ is true where $y \in Y$ and $Y' = Y \setminus \{y\}$.

Note that we assume that all variables of a QBF are quantified, i.e., we are considering closed formulas only. Further, we use standard semantics of conjunction, disjunction, negation, and truth constants. For example, the QBF $\phi_1 = \forall x \exists y . ((x \vee y) \wedge (\neg x \vee \neg y))$ is true, while $\phi_2 = \exists y \forall x . ((x \vee y) \wedge (\neg x \vee \neg y))$ is false. As we see already by this small example, the semantics impose an ordering on the variables w.r.t. the prefix. Given a QBF $\mathcal{Q}.\varphi$, we say that $x <_{\mathcal{Q}} y$ iff x occurs before y in the prefix. If clear from the context, we write $x < y$. In ϕ_1, we have $x < y$, while in ϕ_2, we have $y < x$.

3 Related Work

In practical QBF solving, attempts to parallelize and distribute QBF solvers have a long history (cf. [18] for a survey). Already more than 20 years back, the first distributed QBF solver PQSolve [4] was presented, in a time when QCDCL had not been invented yet. With the advent of QCDCL, several attempts have been made to build parallel QCDCL solvers and implement knowledge-sharing mechanisms for learned clauses and cubes. One example of such a solver is PAQuBE [16]. Unfortunately, the code of most of the early approaches is not available anymore. Following the success of Cube-and-Conquer-based search-space splitting, the QBF solver MPIDepQBF has been presented [14]. While MPIDepQBF does not implement any sophisticated look-ahead mechanisms,

it could demonstrate that even without knowledge-sharing considerable speedup could be achieved. These results serve as motivation for the approach presented in this paper. Unfortunately, MPIDepQBF is implemented in an older version of OCaml that does not run on recent systems and relies on now deprecated libraries, making a comparison impossible. As indicated by its name, it is tailored around the sequential QBF solver DepQBF [17]. Another recent MPI-based QBF solver is HordeQBF [1] which implements knowledge sharing for QCDCL solvers. It is designed in such a way that it allows the integration of any QCDCL solver. In order to integrate a solver, it requires that it implements a certain interface, i.e., programming effort is necessary to add a new solver. To the best of our knowledge, it includes the QBF solver DepQBF only. HordeQBF does not perform search-space splitting, but it is a parallel portfolio solver with clause- and cube sharing. It diversifies the parallel solver instances by different parameter settings. This is different than in sequential portfolio solvers as presented in [12], which select among different solvers based on some properties of the input formula. Overall, a very strong focus on QCDCL-based solvers can be observed for parallel QBF solving frameworks. Because of this, many chances for better solving performance are missed, as nowadays there are many other solvers of orthogonal strength. With ParaQooba we provide a simple way of exploiting the power of the different solving approaches without any integration effort.

4 ParaCooba

Our novel framework ParaQooba (with q in the middle of its name) builds on top of the SAT solver ParaCooba (with c in the middle of its name). In this section, we describe the parts of ParaCooba that are relevant for the remainder of this work for our extension of ParaCooba to ParaQooba.

ParaQooba will be made available publicly during the artifact evaluation under the MIT license, similar to ParaCooba [7,6] which is publicly available on GitHub also under the MIT license[3]. ParaCooba is a distributed Cube-and-Conquer (C&C) solver that implements a proprietary peer-to-peer based load balancing protocol. In contrast to standard D&C solvers the splitting of the search-space can both be done upfront by using a look-ahead solver that produces n cubes or online during solving by lookahead or other heuristics. Amongst other information, the cubes are stored in a binary tree, the *solve tree*.

Solver module. A *solver module* manages the sequential solver that is responsible for solving a subproblem. Different solver modules have different code-bases, but they also generally share common concepts. A solver module implements a parser task, which is created directly after the module was initiated and serves as its starting point. It parses the input formula in its own worker thread and instantiates a solver manager based on the fully parsed formula. The parser task also creates the first solver task as the root of the solve tree.

[3] github.com/maximaximal/Paracooba

Solver Tasks. For ParaCooba, *solver tasks* are paths in the solve tree, whith a *parser task* being used to generate the tree's root. Solver tasks are usually started as children of other tasks, saving references to their parents, with the root solver task being the only exception. A task's depth in the solve tree represents its priority to be worked on: The greater the depth, the more important a task is to be solved locally and the less important it is to be offloaded to other compute nodes by the broker module. Only tasks that were created locally may be distributed.

Broker module. The *broker module* handles relations between solver tasks and processes their results. While the solver module generates tasks, the broker schedules them based on their priorities (their depths) and offloads them if a different compute node has less load than the current node. A task result is propagated upwards across compute nodes, there is no conceptual difference between locally and remotely solved tasks. The broker module is generic and does not rely on a specific solver module, instead providing the environment a solver module works in. It is already provided by ParaCooba and stays the same for different solver modules.

Cube Sources. For generating concrete subproblems, *cube sources* provide assumption literals to leaf solver tasks. A cube source decides whether a given solver task should split again, based on the current configuration (mainly the splitting depth) and the given formula. Every solver module can implement its own cube source, hence there are different kinds of cube sources for different solver modules. On this basis, very flexible mechanisms for the selection of splitting variables can be implemented, ranging from a simple count of literal occurrences to advanced look-ahead heuristics.

Task Tree. The *task tree* built lazily, i.e., only once a leaf is visited, the leaf is either expanded into a sub-tree, or solved. We picture such a tree in Figure 1. This tree has a *depth* of 1, because the path from the tree's root solver task to the leaf solver tasks has a length of 1. Once the active cube source stops further splits from being carried out, the tree's maximum depth is reached. The worker thread currently executing a task then lends a solver instance from the solver manager's central store. Each solver instance is created on-the-fly once (normally initialized based on the parser task) for each worker thread, which can also happen for multiple worker threads in parallel. After a solver instance was created, all other tasks solved by the same worker thread use the same solver instance.

Guiding Paths. The cubes that are given to solver instances as assumptions are called *guiding paths*. They are generated from the path to the leaf being solved. The solver instance then handles the solving internally, blocking the worker thread until either result is generated or the task is terminated. Results are not returned to parents, but instead handled by the broker module, which then traverses the solve tree upwards as far as possible, based on the results already in

the tree. Different kinds of evaluations can be defined on every level using a user-defined *assessment function*. With the result processed by the broker module, the solver task then finishes and the worker thread can take on the next task, based on the next-highest priority. The broker may delete the solver task after it finished processing, if the result was already used somewhere above it in the tree and no information from the original solver task structure is required anymore. Once the broker module has enough information to solve the root task, the result of the formula was computed successfully.

Solver Handle. A *solver handle* wraps instances of a given solver. It must be able to receive an *Assume* event, directly followed by a *Solve* event. While processing these events, a correctly working handle must block its calling thread until a result is found. Additionally, it must be fully re-entrant after finishing processing, so that the next solver task can apply new assumptions. On top of this, a handle must also be able to process a *Terminate* event, stopping the solver and early-returning control to its calling thread. Such a termination event may happen at any time, as it is generated by other solver tasks. This possibility of random terminations was an issue for our extension to ParaQooba, as it complicated synchronization of all involved threads.

QBF Solver Module. ParaCooba already provided a basic *QBF solver module* similar to the approach seen in MPIDepQBF. It implemented a QDIMACS-parser in a new solver module based on the SAT module. It realizes a simple cube source that returns the variable at the nth position in the prefix, with n being the current depth of a solver task. The solve tree is built using two adapted assessment functions: one for variables quantified $\forall$ (requiring all sub-trees to be true), one for $\exists$ (requiring at least one sub-tree to be true). The assessment functions also use ParaCooba's cancellation-support to terminate unneeded siblings after results already satisfy the respective subproblem. As backend solver, it exclusively uses DepQBF that provides an incremental API (which no other recent solver provides, to the best of our knowledge).

Summary. With its already existing tree-based QBF solving module together with its support for distributed solving, ParaCooba provides a stable basis for building an advanced parallel QBF solver. While the existing QBF module is rather uncompetitive with a few exceptions that indicate its potential, its core infrastructure turned out to be very useful to build our novel framework ParaQooba that offers built-in portfolio support.

The networking support mentioned above enables combining multiple compute nodes by giving each peer a connection to the main node. This is achieved with setting the `--known-remote` option. With this feature it becomes possible to easily distribute larger problem instances on a cluster or in the cloud.

5 Architecture of ParaQooba: Combining Divide-and-Conquer Portfolio Solving

Our framework ParaQooba combines Divide-and-Conquer (D&C) search space splitting with portfolio solving. The key feature of ParaQooba compared to ParaCooba is to allow portfolio solving at different search depths. The idea is illustrated in Figure 1. Both approaches are widely used to realize parallel and distributed SAT and QBF solvers. The D&C approach has been especially successful for hard combinatorial SAT problems [11] in a variant called Cube-and-Conquer (C&C). The C&C approach relies on powerful, but expensive lookahead solvers that heuristically decide which variables shall be considered for splitting. In its original SAT version, ParaCooba builds upon this idea [7].

For a QBF $Q_1XQ_2Y\mathcal{Q}.\varphi$ with $Q_1 \neq Q_2$ and $Q_1, Q_2 \in \{\forall, \exists\}$ though, the possible choices for variable selection are more restricted because of the quantifier prefix. In general, only variables from the outermost quantifier block Q_1X may be considered, because otherwise, the value of the formula might change. Jordan et al. [14] observed that for QBF following the sequential order of the variables in the first quantifier block already leads to improvements compared to the sequential implementation of DepQBF. The already existing QBF solver module of ParaCooba (see section 4) relied on this observation: it traverses the prefix of a PCNF and splits each visited leaf into two sub-trees, respecting both universal and existential quantifiers, until a pre-defined maximum depth is reached. Hence, it re-implements the approach of MPIDepQBF in ParaCooba.

Our framework ParaQooba generalizes the previous QBF module of ParaCooba not only by generalizing the interface in such a manner that any QBF solver can be easily (without programming effort) integrated as backend solver. Now it is also possible to run several solvers in the leaves as shown in Figure 2 for one split. Overall, ParaQooba realizes the following approach. The search-space is split according to the variable ordering of the prefix until a given depth. Once one of the sub-trees of an existentially quantified variable split is found to be true, the other sibling is terminated. Only when both siblings return false, the whole split returns false. Universal splits work in a dual manner: the result is only true if both sub-trees are found to be true and false otherwise. This property of QBF enables efficient termination of sub-tasks.

In ParaQooba, we now also parallelize each solver call over several QBF solvers with orthogonal strategies. Compared to prior approaches [18], we run a portfolio of multiple solvers in the leaves of the solve tree instead of only parallelizing its root. Having just one tree leads to several advantages: We are more flexible and may also call a preprocessor (e.g. Bloqqer) before each solve call. We also only instantiate the tree once, saving memory and enabling early-termination of sibling solver tasks.

6 Implementation

This section describes the extension of the SAT solver ParaCooba (for an overview see section 4) to our QBF solving framework ParaQooba. As Para-

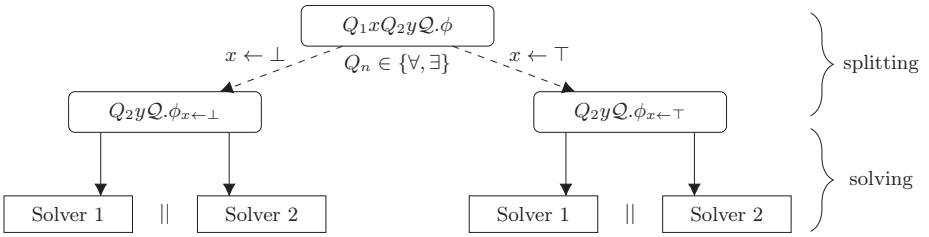


Fig. 1: Divide-and-Conquer with arbitrary-many levels of splitting and sub-formulas on the leaves solved by a portfolio of different sequential solvers

Cooba was originally not designed for portfolio support, several modifications and extensions were necessary. To this end, we first present the new QBF module of ParaQooba followed by a discussion of novel search-space pruning facilities.

6.1 The ParaQooba QBF Module

We generalized the already existing QBF solver handle to become an abstract base class, which now can be either a single solver handle or a *portfolio handle*. The latter unifies multiple handles into one, emulating a blocking and re-entrant interface. Once a portfolio handle is initialized, it starts one thread per internally wrapped handle. Each such thread implements a small state machine, waiting for events on a shared queue. Once the portfolio handle receives an assumption (a temporary truth assignment of a variable for one solver call), it is forwarded to all internal threads and is worked on by each wrapped solver in parallel.

If a portfolio handle was terminated before a solve call was issued, the internal handles would enter an invalid state. To circumvent this situation, an assumption event also directly triggers the internal state machine to continue into the solve state. Once the solve request actually arrives, it is just translated to an empty event, which, after it finished processing, indicates that a result was computed. A termination event is forwarded to the internal solver handles, but is limited to only one event per solve cycle.

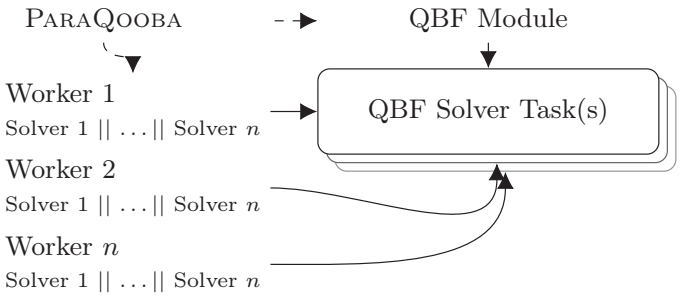


Fig. 2: The ParaQooba framework

The first internal solver handle to compute a result returns and sends a termination event to all sibling solvers. The result is saved and the portfolio handle waits for all internal handles to be ready to receive the next assumption, i.e., returning all solvers to a known state. Once every internal handle has reached that, the portfolio handle finally returns to its calling thread, forwarding the result of the inner handle. Because of thread scheduling and fast solving of trivial subproblems, a result can be forwarded even before the other sibling has been started, letting the broker module already complete a task before it itself has created both child tasks. This effect lead to some issues and had to be mitigated by adding some conditions on a task already being terminated even though it did not yet run to completion. Because a task will only be scheduled after the initial call to its assessment function, not many such checks were needed.

As many QBF solvers lack APIs, we have to work with their binaries that generally only read QDIMACS files. For this, we use the QuAPI interfacing library, that adds well-performing assumption-based reasoning support to generic solver binaries [9]. By not relying on specialized modifications of a solver's source code, we are able to plug-in generic third-party solvers, completely composable at runtime. Our ParaQooba module provides the `--quapisolver` parameter, that either directly specifies the leaf solver to be used, or automatically generates a portfolio handle to wrap multiple parallel leaf solvers. Note that our approach works for QBFs starting with existential as well as with universal quantification.

In its standard configuration, ParaQooba returns whether a given instance is found to be true or false. When enabling trace output using `-t`, it also supports printing the specific solver and the subproblem (including its guiding path) that produced a result. Using this machinery, one obtains an environment to experiment with benchmarks and to see how multiple solvers complement each other for the generated sub-formulas. The trace output is also useful when fully expanding a QBF formula by specifying a tree-depth of -1. While not advised for any real formulas, this was a well-received debugging aid for stress-testing new features. The opposite to this can also be done, by applying a tree-depth of 0. This directly solves the root task, without splitting the formula. This was also how the configuration PQ Portfolio with depth 0 (as discussed in the experimental evaluation below) was executed.

6.2 Search-Space Pruning

Preprocessing in the leaves. We modified the QBF preprocessor Bloqqer to allow forwarding output directly into a given solver binary by adding a `-p` argument. Internally, this writes the complete formula with added assumptions into the standard input of Bloqqer's preprocessing pipeline.

To plug e.g. Caqe into such a processing chain and then into ParaQooba, one may use our QBF solver module's command line option `--quapisolver bloqqer-popen@-p=caqe`. Deferring preprocessing until solving the leaves preserves the original formula structure of a formula during the split phase. We discuss the effects of this later in subsection 7.4.

Integer-Split Reduction. In many planning and verification encodings, the variables of a quantifier block QX are interpreted as bitvectors representing m nodes of a graph. Assume that $n = |X|$ bits with $m \leq 2^n$ are used for modeling the states of the graph. Then $2^n - m$ assignments to X are not relevant, but as a solver is agnostic of this information, it has to consider all assignments.

If m is known to the user, ParaQooba can be called with the option `--intsplit` (once or multiple times, once for each layer). One integer-split is counted as one layer in the task tree, so a tree-depth of two would split another quantifier into two more tasks for each state encoded in the previous integer-based split. To provide an example: Setting `--intsplit 5` creates 5 child-tasks in the task tree, spanning over the first $\lceil \log_2 5 \rceil = 3$ boolean variables from the quantifier prefix. When not using doing an integer-based split, these 3 variables would have to be expanded over 3 layers in the task tree, each inner task being split into two child tasks, resulting in 8 leaves , opposed to the 5 from before. Thus, integer-based splits require less intermediate splitting tasks to model the same formula, reducing the work to be done by the load-balancing mechanism in the Broker module. These integer splits are efficiently distributed over the network by relying on both the config-system and an extended QBF cube source. The cube source always saves the current guiding path, applying new splits, and in turn new assumptions, by appending to that path. The cube source itself is automatically serialized when a task is chosen to be offloaded to another compute node. While the possible savings are large, one has to exert great caution when using this feature, as it might change the semantics of a formula.

7 Evaluation

In this section, we evaluate ParaQooba on recent benchmarks and compare it to (sequential) state-of-the-art QBF solvers. As sequential backend solvers, we use the latest versions of DepQBF [17] as QCDCL solver, Caqe [23] as clausal-abstraction solver, and RaReQs [13] as recursive abstraction refinement solver. For preprocessing, we use Bloqqer [3] (version 31). All of these solvers were top-ranked in the most recent edition of QBFEval'22 [22]. For our experiments we used the benchmarks of the PCNF-track of this competition. The main questions we want to answer with our evaluation are as follows:

- how does the parallel portfolio-leaf approach of ParaQooba perform in comparison to the individual sequential solvers?
- how does the parallel portfolio-leaf approach of ParaQooba perform in comparison to the virtual portfolio solver of the sequential solvers?
- what is the impact of performing the preprocessing in the leaves instead on the original input formula?

We ran our experiments on machines with dual-socket 16 core AMD EPYC 7313 processors with 3.7 GHz sustained boost clock speed and 256 GB main memory. Each task was assigned as many physical cores as its setup required, except for tasks with more than 32 concurrent threads, which were exclusively

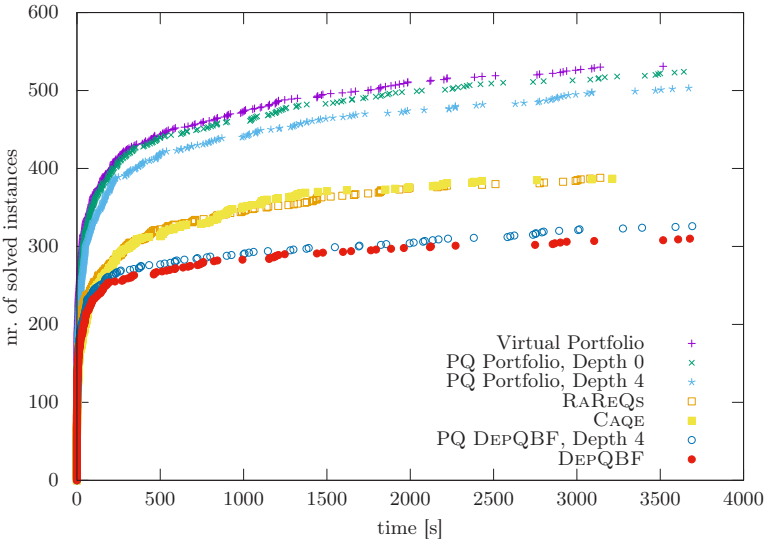


Fig. 3: Full summary of all solved instances with all different solvers without preprocessing. While Divide-and-Conquer (Depth 4) formulas solves 33 instances that no sequential solver solved, it solves 28 instances less in total.

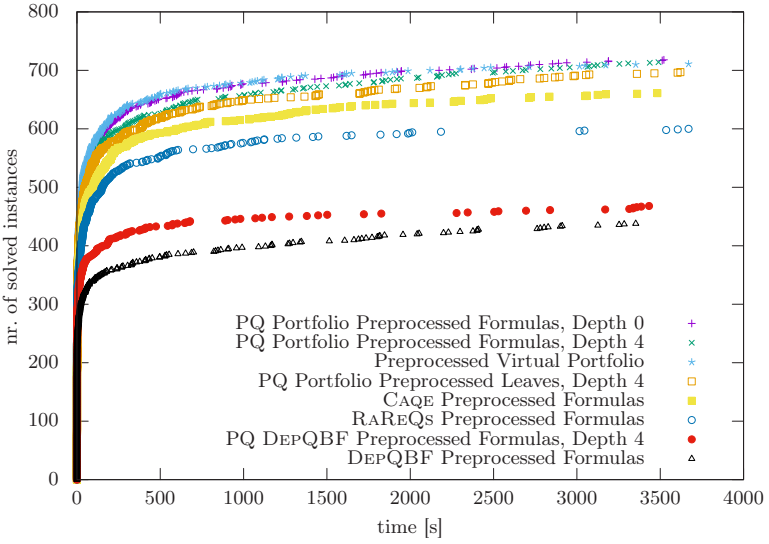


Fig. 4: Full summary of all solved instances with all different solvers with Bloqqer preprocessing. PQ Portfolio (Depth 4) solves 45 instances no sequential solver could solve and solves 3 more in total.

assigned a whole node each as to not be slowed down by other loads. The effects of over-committing in case of three concurrent portfolio solvers (48 threads running in parallel with only 32 physical cores available) are discussed below in subsection 7.3.

Please note that in this evaluation we do not use the networking features provided by ParaCooba, as we focus on applicability to QBF and not on the already presented scalability of the networking component (for the details see [3]).

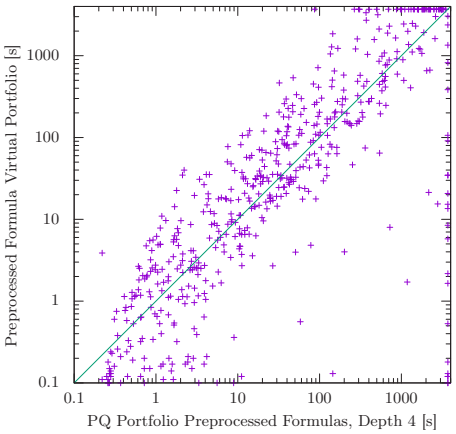
7.1 Overall Performance Comparison

In order to exploit our hardware with 32 physical cores and 64 logical cores in the best possible way, we mainly focus on a *splitting depth of four* in the following. With this depth, 16 worker threads are generated for each problem and with three sequential backend solvers, overall 48 processes are started. We call this configuration *PQ Portfolio, Depth 4*. For understanding the impact of splitting, we also consider other depths as well. With *PQ Portfolio, Depth 0* we refer to the configuration in which splitting is disabled. This configuration is particularly interesting, because compared to the virtual best solver (VBS), it reveals the overhead introduced by our framework (see also the discussion below). In order to show the improvements of ParaQooba compared to the QBF module without portfolio solving that was already available in ParaCooba [6], we also included the configuration *PQ DepQBF, Depth 4*.

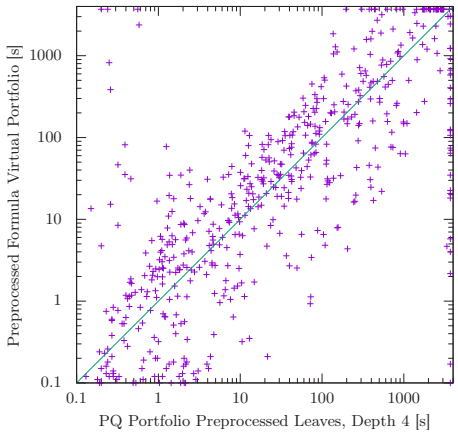
Figure 3 shows the overall results of our evaluation *without preprocessing*. Both configurations of ParaQooba, *PQ Portfolio, Depth 0* and *PQ Portfolio, Depth 4* are considerably better than the single sequential solvers as well as the basic non-portfolio QBF module of ParaCooba only solving with DepQBF (PQ DepQBF, Depth 4). However, compared to the virtual portfolio, 28 instances less are solved in total (for an explanation see below). On the positive side, 33 formulas can be solved by our new approach that could not be solved by any sequential solver. The situation changes when preprocessing is applied (cf. Figure 4). Now ParaQooba in configuration *PQ Portfolio Preprocessed Formulas, Depth 4* is able to solve most formulas. It even solves more formulas than the *Preprocessed Virtual Portfolio*, indicating the potential of our approach.

A detailed analysis is given in Figure 5. By comparing the number of solved instances to the solve time of individual (preprocessed) problem instances, we see a small average speedup when using ParaQooba with depth 4 compared to a virtual portfolio solver in Figure 5a. The more trivial instances tend to be solved quicker using a sequential solver, while the harder to solve instances tend to be solved faster with the Divide-and-Conquer approach of ParaQooba.

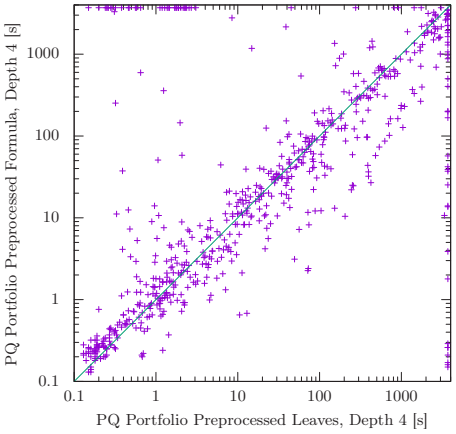
Next, we used the preprocessed leaves functionality introduced in subsection 6.2. Here ParaQooba generates its guiding paths using the original formula and applies Bloqqer only in the leaves of the solve tree. In this configuration, some problem instances take longer to solve than when preprocessing the full formula, while others can be solved quicker. We present these results in Figure 5b. Such a result was expected, as it is conceptually similar to inprocessing.



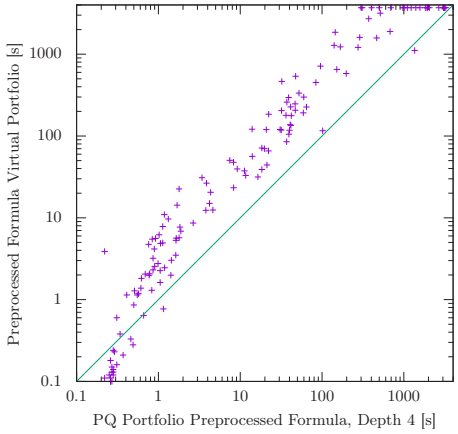
(a) ParaQooba with Depth 4 compared to Virtual Portfolio



(b) Same as a, but with Bloqqer preprocessing in leaves



(c) ParaQooba: preprocessing of leaf formulas compared to preprocessing of input formula



(d) ParaQooba with Depth 4 compared to Virtual Portfolio on Hex formulas

Fig. 5: Detailed comparison of ParaQooba against the virtual portfolio of DepQBF, Caqe, and RaReQs in a, b, d. In a, ParaQooba solves 45 instances that no sequential solver could solve. In b, ParaQooba solves 38 instances no sequential solver could solve, 8 of which also could not be solved with portfolio over preprocessed formulas as in a. d focuses only on preprocessed formulas from the Hex benchmark family. In c, we directly compare preprocessing in the leaves to preprocessing in the input formula.

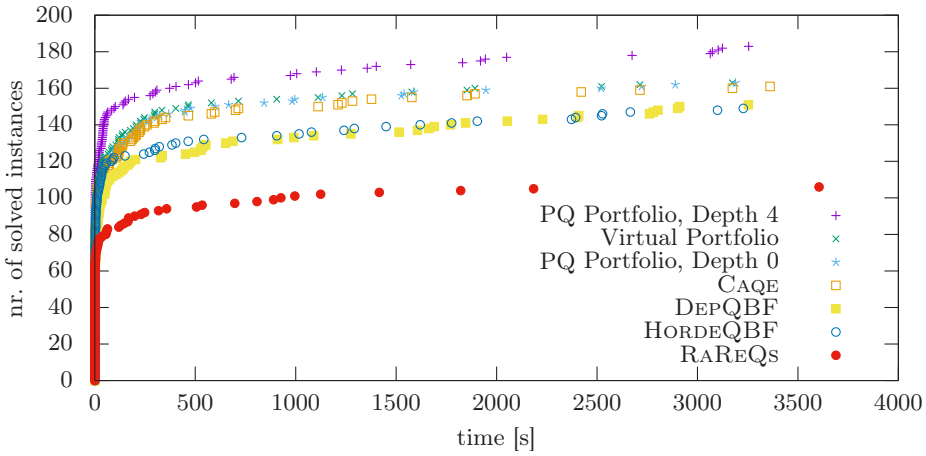


Fig. 6: Preprocessed formulas of the Hex positional game planning [20,25] benchmarks from the QBF22 benchmark set. Also compared to HordeQBF [1] as available state-of-the-art parallel QBF solver.

When considering the formulas that were exclusively solved by ParaQooba, then the variant with preprocessing the full formula up-front performed best followed by the variant with preprocessing in the leaves. These formulas include verification and synthesis benchmarks with 2–3 quantifier alternations as well as many encodings of the game Hex with 13, 15 or 17 quantifier alternations. Table 1 in the appendix lists all instances (48) that were only solved with some variant of ParaQooba. It also lists which variant was the fastest.

7.2 Family-Based Analysis

To understand which formula families benefit most from our Divide-and-Conquer solving strategy, we compared the (wall-clock) solve time of ParaQooba to the virtual portfolio solver. We calculated the speedup by dividing the solve time of the sequential solver by the solve time of ParaQooba. The instances with the highest speedups were some reachability queries (up to 18.09), the Hex game planning family (17.64), multipliers (16.46), and the formula_add family (15.16). More detailed results are appended in Table 2. Together with the number of Hex instances only ParaQooba solved (21), this makes Hex game planning the benchmark family with the best overall results in our evaluation. A comparison between ParaQooba and other solvers is shown in Figure 6.

7.3 Scalability of our Approach

As already discussed above, using 16 workers leads to overcommitting cores when solving with a portfolio of more than two solvers. To quantify this, we did

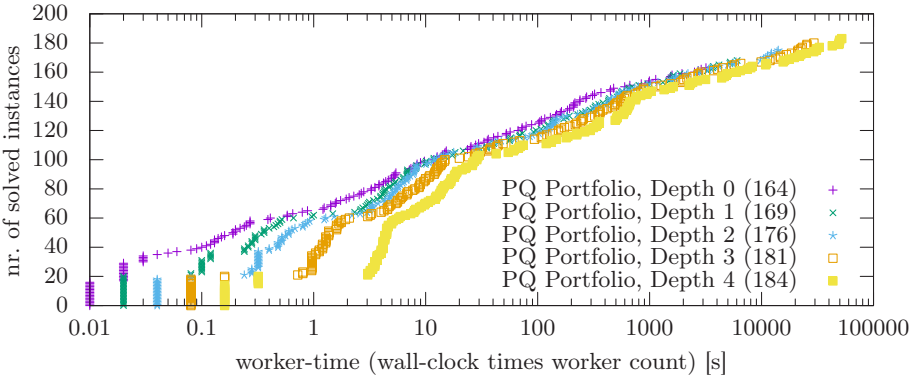


Fig. 7: Hex Scalability with preprocessed formulas. Depth 4 suffers from over-committing the available CPU-cores on our hardware and is relatively slow for the first few problems, but still solves more instances overall.

a scalability experiment with different worker counts. Because the Hex planning benchmarks had the most predictable performance, we focused this experiment on these formulas. Figure 7 shows the scalability graph, where the X-axis has been multiplied by the number of workers used, to visualize the cost of increased CPU-time compared to reduced wall-clock solve time. The impact of over-committing CPU cores can be clearly observed in the results of the portfolio with depth 4. This curve solves more compared to the others and takes longer to solve the first 140 instances, until the curves become more similar again.

7.4 Preprocessed Leaves compared to Preprocessed Formulas

We compared preprocessing the whole formula at once using BLOQQER to calling BLOQQER using `bloqqer-popen` in each leaf after first splitting on the unchanged formula. The first variant modifies the original prefix, including the quantifier ordering. Because the used splitting algorithm generates guiding paths by following this quantifier ordering, the different approaches lead to vastly different results. Figure 5c visualizes these differences by scattering both variants together.

Looking at the specific benchmarks benefiting from the two variants, we often observed improvements to one variant per family. This strongly suggests that adaptive preprocessing and inprocessing techniques could further improve solving performance, even without otherwise changing solvers themselves.

7.5 Lessons Learned

One would expect that for any given problem, parallel portfolio solvers are as fast as the fastest used solver. While this statement is conceptually true, we encountered some formulas where PQ-Portfolio gave comparatively bad results, while a solver alone could solve the same formula quicker or even instantly.

We investigated this in more detail and found several segmentation faults in Caqe and API inconsistencies in DepQBF that were encountered because of some corner-case structures of the generated subproblems (e.g., by enforcing the values of certain variables). We reported these issues to the solver developers and hope to obtain fixes soon. Having this issues fixed would lead to a more performant general solution and to a more robust user experience. In sequential execution of these solvers, we did not encounter any problems on the unmodified competition benchmarks without added unit clauses.

Currently, we adopt the following work-around. Segmentation faults of the sequential solvers are handled in our QBF module using the indirection provided by QuAPI. Once an unrecoverable error occurs in the solver child process, it exits and returns the error up through QuAPI's factory process and into the solver handle. There, such a result is interpreted as *Unknown*, which is invalid and therefore ignored, letting the portfolio wait for other results. We provide all affected formulas that we found in the artifact submitted alongside this paper.

We also observed that calling a solver via its API might lead to a considerably different behavior than calling a solver from the command line, i.e., different optimizations are activated when calling a solver through its API compared to using the command-line binary. Such behavior can be mitigated by not using the API directly, and instead relying on QuAPI, even if an API would be available. This fixes the issues with DepQBF, which solves some formulas (with assumptions supplied as unit clauses) in under one second if used as a solver binary, but not when applying assumptions through its API. We also supply all found formulas that triggered this issue in the submitted artifact.

8 Conclusions

We presented ParaQooba, a parallel and distributed QBF solving framework that combines search-space splitting with portfolio solving. We designed the framework in such a way that any sequential QBF solver binary can be easily integrated without any implementation effort. Our experiments demonstrate that this approach in combination with sequential preprocessing lead to considerable performance improvements for certain formula families.

With our framework, we provide a stable infrastructure that has the potential for many future extensions. For example, we did not incorporate any advanced splitting heuristics as in modern Cube-and-Conquer solvers. We expect that with more advanced heuristics, combined with adaptive but possibly non-deterministic re-splitting of leaves, even more speedups could be achieved.

In addition to the presented experiments, we also evaluated the novel integer-split feature (cf. subsection 6.2) with the Hex benchmark family. By providing the number of valid game states to ParaQooba, we could increase the splitting depth as well as the number of solved instances. We see much potential of providing encoding-specific or domain-specific knowledge to the solver and will investigate this in future work.

Data Availability Statement

Data used for benchmarking the described software, including source code, are made available permanently under a permissive license in a public artifact on Zenodo. Raw source data for the figures presented in this paper are also included [8].

A Instances Only Solved by ParaQooba

Name	Clauses	Variables	QA	Time [s]	Res	Variant
b21_C_3_206	242896	3270	3	265.77	$\top$	full
c1_Debug_s3_f1_e1_v1	1775758	379113	3	3164.34	$\top$	full
c2_Debug_s3_f1_e1_v2	431970	98425	3	1834.27	$\top$	full
cache-coherence-2-fixpoint-2	10648	3686	2	0.56	$\bot$	leaves
cmu.dme1.B-f3	4540	1795	3	0.2	$\top$	leaves
cmu.dme2.B-f3	6151	2342	3	818.3	$\top$	leaves
LoginService	21667	5289	2	1086.07	$\bot$	orig
query64_query42_1344n	3423	1426	2	86.73	$\top$	full
hex_compact_goal_witness_based_hein_03_6x6-13.pg	3401	1056	15	2594.27	$\bot$	leaves
hex_compact_goal_witness_based_hein_05_6x6-13.pg	3493	1071	15	3102.97	$\top$	full
hex_compact_goal_witness_based_hein_17_6x6-13.pg	3430	1060	15	1919.64	$\bot$	full
hex_compact_goal_witness_based_hein_18_7x7-13.pg	4256	1267	15	1401.12	$\bot$	full
hex_compact_goal_witness_based_hein_02_5x5-13.pg	3134	1007	15	308.99	$\top$	full
hex_compact_goal_witness_based_hein_15_5x5-15.pg	3667	1195	17	3063.67	$\top$	full
hex_symbolic_explicit_goal_hein_03_6x6-11.pg	3421	902	13	693.11	$\bot$	full
hex_symbolic_explicit_goal_hein_05_6x6-11.pg	3611	918	13	501.29	$\bot$	full
hex_symbolic_explicit_goal_hein_18_7x7-11.pg	3084	1021	13	447.7	$\bot$	leaves
hex_symbolic_explicit_goal_hein_02_5x5-11.pg	2480	739	13	973.33	$\bot$	full
hex_symbolic_explicit_goal_hein_16_5x5-11.pg	2376	731	13	301.31	$\bot$	full
hex_symbolic_implicit_goal_hein_03_6x6-13.pg	3069	1001	15	1830.57	$\bot$	full
hex_symbolic_implicit_goal_hein_17_6x6-13.pg	3097	1005	15	2674.38	$\bot$	full

hex_symbolic_implicit_goal_hein_02_5x5-13.pg	2812	952	15	404.36	⊤	full
hex_symbolic_implicit_goal_hein_15_5x5-15.pg	3106	1072	17	1944.27	⊤	full
hex_witness_based_hein_03_6x6-13.pg	7174	1917	13	2050.04	⊥	full
hex_witness_based_hein_05_6x6-13.pg	7456	1962	13	1005.06	⊤	full
hex_witness_based_hein_17_6x6-13.pg	7353	1936	13	1572.7	⊥	full
hex_witness_based_hein_18_7x7-13.pg	9577	2405	13	1102.69	⊥	full
hex_witness_based_hein_20_6x6-13.pg	7551	1962	13	3123.99	⊥	full
hex_witness_based_hein_15_5x5-15.pg	7423	2136	15	2489.7	⊤	leaves
OrgSynth_mitexams_p02_l_6	83500	23384	3	1852.22	⊤	full
OrgSynth_mitexams_p02_l_7	97214	27239	3	2693.19	⊤	full
OrgSynth_mitexams_p03_l_5	106413	29730	3	2897.47	⊤	full
OrgSynth_mitexams_p07_l_5	165039	46587	3	2469.04	⊥	leaves
OrgSynth_mitexams_p16_l_6	53448	15692	3	2169.18	⊤	full
OrgSynth_mitexams_p16_l_7	62141	18265	3	3054.75	⊤	leaves
OrgSynth_mitexams_p19_l_6	106252	29346	3	3489.44	⊤	full
OrgSynth_mitexams_p20_l_7	74375	21534	3	1782.51	⊥	full
OrgSynth_mitexams_p01_l_4	65294	17864	3	1609.48	⊥	full
OrgSynth_mitexams_p05_l_3	79279	22897	3	2055.46	⊤	leaves
OrgSynth_mitexams_p05_l_4	105042	30409	3	2253.59	⊤	full
OrgSynth_mitexams_p10_l_3	44309	12864	3	870.16	⊤	full
OrgSynth_mitexams_p10_l_4	58490	17046	3	2163.5	⊤	full
OrgSynth_mitexams_p13_l_3	52653	14953	3	1310.32	⊤	full
OrgSynth_mitexams_p13_l_4	69554	19819	3	2592.6	⊤	leaves
OrgSynth_sat18_p09_l_3	52653	14953	3	1765.8	⊤	leaves
OrgSynth_sat18_p09_l_4	69554	19819	3	2328.99	⊤	leaves
OrgSynth_sat18_p11_l_4	85537	23860	3	2123.52	⊥	leaves
OrgSynth_sat18_p12_l_4	82734	23155	3	2803.72	⊥	leaves

Table 1: 48 instances that were only solved by a ParaQooba configuration. QA: Quantifier Alternations, Res: Result, Variant: ParaQooba configuration that solved the problem the fastest (preprocess full formula, preprocess leaves, original formula).

B Instances Solved faster by ParaQooba

Name	PQ [s]	VPS [s]	Speedup	Res
nreachq_query71_1344n	2.21	39.97	18.09	⊥
hex_witness_based_hein_08_5x5-11.pg	0.22	3.88	17.64	⊤
mult9.sat	2.11	34.73	16.46	⊤
add5_COMPLETE	1.78	26.98	15.16	⊤
hex_symbolic_explicit_goal_hein_10_5x5-11.pg	32.23	465.43	14.44	⊥
hex_compact_goal_witness_based_hein_10_5x5-13.pg	144.98	1853.09	12.78	⊤
hex_symbolic_explicit_goal_hein_11_5x5-09.pg	1.79	22.53	12.59	⊥
hex_symbolic_implicit_goal_hein_03_6x6-11.pg	47.52	538.03	11.32	⊥
reachqu_query60_1344n	7.57	77.4	10.22	⊥
query71_query36_1344n	11.38	105.83	9.30	⊥
hex_symbolic_explicit_goal_hein_08_5x5-09.pg	1.18	10.94	9.27	⊥
hex_symbolic_implicit_goal_hein_20_6x6-11.pg	140.49	1282.38	9.13	⊥
hex_witness_based_hein_06_4x4-11.pg	3.41	30.9	9.06	⊥
hex_compact_goal_witness_based_hein_10_5x5-11.pg	13.97	121.04	8.66	⊥
hex_symbolic_implicit_goal_hein_19_5x5-11.pg	1.69	14.29	8.46	⊤
hex_symbolic_implicit_goal_hein_16_5x5-11.pg	22.26	184.75	8.30	⊥
sortnetsort10.AE.stepl.008	13.33	107.07	8.03	⊥
add7_REDUCED	135.58	1051.44	7.76	⊤
reachqu_query64_1344n	128.4	982.54	7.65	⊥
hex_compact_goal_witness_based_hein_02_5x5-11.pg	39.04	295.57	7.57	⊥
amba4b9y.unsat	10.9	81.72	7.50	⊥
hex_symbolic_implicit_goal_hein_15_5x5-13.pg	95.67	714.78	7.47	⊥
hex_compact_goal_witness_based_hein_15_5x5-13.pg	167.18	1229.74	7.36	⊥
hex_symbolic_implicit_goal_hein_06_4x4-11.pg	1.32	9.67	7.33	⊥
hex_compact_goal_witness_based_hein_16_5x5-13.pg	372.26	2713.59	7.29	⊤

Table 2: Instances that ParaQooba (PQ) solved faster compared to a virtual portfolio solver (VPS) that also solved the same problem, ordered by the relative speedup and limited to the top 25 entries. Res: Result, Speedup: $\frac{\text{VPS}[s]}{\text{PQ}[s]}$.

References

1. Balyo, T., Lonsing, F.: HordeQBF: A modular and massively parallel QBF solver. In: Creignou, N., Berre, D.L. (eds.) Proc. of the 19th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer

Science, vol. 9710, pp. 531–538. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_33
2. Beyersdorff, O., Janota, M., Lonsing, F., Seidl, M.: Quantified boolean formulas. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 1177–1221. IOS Press (2021). https://doi.org/10.3233/FAIA201015
3. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N.S., Sofronie-Stokkermans, V. (eds.) Proc. of the 23rd Int. Conf. on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 6803, pp. 101–115. Springer (2011). https://doi.org/10.1007/978-3-642-22438-6_10
4. Feldmann, R., Monien, B., Schamberger, S.: A distributed algorithm to evaluate quantified boolean formulae. In: Kautz, H.A., Porter, B.W. (eds.) Proc. of the 17th Nat. Conf. on Artificial Intelligence and 12th Conf. on on Innovative Applications of Artificial Intelligence (AAAI/IAAI). pp. 285–290. AAAI Press / The MIT Press (2000), http://www.aaai.org/Library/AAAI/2000/aaai00-044.php
5. Frioux, L.L., Baarir, S., Sopena, J., Kordon, F.: Modular and efficient divide-and-conquer SAT solver on top of the painless framework. In: Vojnar, T., Zhang, L. (eds.) Proc. of the 25th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 11427, pp. 135–151. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_8
6. Heisinger, M.: Distributed SAT & QBF solving: The paracooba framework. Master Thesis, JKU Linz (2021)
7. Heisinger, M., Fleury, M., Biere, A.: Distributed cube and conquer with paracooba. In: Pulina, L., Seidl, M. (eds.) Proc. of the 23rd Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 12178, pp. 114–122. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_9
8. Heisinger, M., Seidl, M., Biere, A.: Artifact for Paper ParaQooba: A Fast and Flexible Framework for Parallel and Distributed QBF Solving (Nov 2022). https://doi.org/10.5281/zenodo.7554207
9. Heisinger, M., Seidl, M., Biere, A.: QuAPI: Adding assumptions to non-assuming SAT & QBF solvers. In: Konev, B., Schon, C., Steen, A. (eds.) Proc. of the Workshop on Practical Aspects of Automated Reasoning (FLoC/IJCAR). CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022), http://ceur-ws.org/Vol-3201/paper1.pdf
10. Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause elimination for SAT and QSAT. J. Artif. Intell. Res. **53**, 127–168 (2015). https://doi.org/10.1613/jair.4694
11. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Proc. of the 7th Int. Conf. on Hardware and Software: Verification and Testing (HVC). Lecture Notes in Computer Science, vol. 7261, pp. 50–65. Springer (2011). https://doi.org/10.1007/978-3-642-34188-5_8
12. Hoos, H.H., Peitl, T., Slivovsky, F., Szeider, S.: Portfolio-based algorithm selection for circuit QBFs. In: Hooker, J.N. (ed.) Proc. of the 24th Int. Conf. on Principles and Practice of Constraint Programming (CP). Lecture Notes in Computer Science, vol. 11008, pp. 195–209. Springer (2018). https://doi.org/10.1007/978-3-319-98334-9_13
13. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) Proc. of

the 15th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 7317, pp. 114–128. Springer (2012). https://doi.org/10.1007/978-3-642-31612-8_10
14. Jordan, C., Kaiser, L., Lonsing, F., Seidl, M.: MPIDepQBF: Towards parallel QBF solving without knowledge sharing. In: Sinz, C., Egly, U. (eds.) Proc. of the 17th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 8561, pp. 430–437. Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_32
15. Kaufmann, D., Kauers, M., Biere, A., Cok, D.: Arithmetic verification problems submitted to the SAT Race 2019. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Race 2019 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2019-1, p. 49. University of Helsinki (2019)
16. Lewis, M., Schubert, T., Becker, B., Marin, P., Narizzano, M., Giunchiglia, E.: Parallel QBF solving with advanced knowledge sharing. Fundam. Informaticae **107**(2-3), 139–166 (2011). https://doi.org/10.3233/FI-2011-398
17. Lonsing, F., Egly, U.: DepQBF 6.0: A search-based QBF solver beyond traditional QCDCL. In: de Moura, L. (ed.) Proc. of the 26th Int. Conf. on Automated Deduction (CADE). Lecture Notes in Computer Science, vol. 10395, pp. 371–384. Springer (2017). https://doi.org/10.1007/978-3-319-63046-5_23
18. Lonsing, F., Seidl, M.: Parallel solving of quantified boolean formulas. In: Hamadi, Y., Sais, L. (eds.) Handbook of Parallel Constraint Reasoning, pp. 101–139. Springer (2018). https://doi.org/10.1007/978-3-319-63516-3_4
19. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 133–182. IOS Press (2021). https://doi.org/10.3233/FAIA200987
20. Mayer-Eichberger, V., Saffidine, A.: Positional games and QBF: The corrective encoding. In: Pulina, L., Seidl, M. (eds.) Proc. of the 23rd Int. Conf. on Theory and Applications of Satisfiability Testing (SAT). Lecture Notes in Computer Science, vol. 12178, pp. 447–463. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_31
21. Pulina, L., Seidl, M.: The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). Artif. Intell. **274**, 224–248 (2019). https://doi.org/10.1016/j.artint.2019.04.002
22. Pulina, L., Seidl, M., Shukla, A.: QBFEval 2022. http://www.qbflib.org/qbfeval22.php (2022)
23. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Kaivola, R., Wahl, T. (eds.) Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD). pp. 136–143. IEEE (2015)
24. Sanders, P., Schreiber, D.: Mallob: Scalable SAT solving on demand with decentralized job scheduling. J. Open Source Softw. **7**(77), 4591 (2022). https://doi.org/10.21105/joss.04591
25. Shaik, I., Mayer-Eichberger, V., van de Pol, J., Saffidine, A.: Implicit state and goals in QBF encodings for positional games (extended version) (2023). https://doi.org/10.48550/ARXIV.2301.07345
26. Shukla, A., Biere, A., Pulina, L., Seidl, M.: A survey on applications of quantified boolean formulas. In: Proc. of the 31st IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI). pp. 78–84. IEEE (2019). https://doi.org/10.1109/ICTAI.2019.00020

Inferring Needless Write Memory Accesses on Ethereum Bytecode*

Elvira Albert[1], Jesús Correas[1], Pablo Gordillo[1](✉), Guillermo Román-Díez[2], and Albert Rubio[1]

[1] Complutense University of Madrid, Madrid, Spain
pabgordi@ucm.es
[2] Universidad Politécnica de Madrid, Madrid, Spain

Abstract. Efficiency is a fundamental property of any type of program, but it is even more so in the context of the programs executing on the blockchain (known as *smart contracts*). This is because optimizing smart contracts has direct consequences on reducing the costs of deploying and executing the contracts, as there are fees to pay related to their bytes-size and to their resource consumption (called *gas*). Optimizing memory usage is considered a challenging problem that, among other things, requires a precise inference of the memory locations being accessed. This is also the case for the Ethereum Virtual Machine (EVM) bytecode generated by the most-widely used compiler, `solc`, whose rather unconventional and low-level memory usage challenges automated reasoning. This paper presents a static analysis, developed at the level of the EVM bytecode generated by `solc`, that infers write memory accesses that are needless and thus can be safely removed. The application of our implementation on more than 19,000 real smart contracts has detected about 6,200 needless write accesses in less than 4 hours. Interestingly, many of these writes were involved in memory usage patterns generated by `solc` that can be greatly optimized by removing entire blocks of bytecodes. To the best of our knowledge, existing optimization tools cannot infer such needless write accesses, and hence cannot detect these inefficiencies that affect both the deployment and the execution costs of Ethereum smart contracts.

1 Introduction

EVM and memory model. Ethereum [27] is considered the world-leading programmable blockchain today. It provides a virtual machine, named EVM (Ethereum Virtual Machine) [21], to execute the programs that run on the blockchain. Such programs, known as Ethereum "smart contracts", can be written in high-level programming languages such as Solidity [6], Vyper [4], Serpent [3] or Bamboo [1] and they are then compiled to EVM bytecode. The EVM bytecode is the code finally deployed in the blockchain, and has become a uniform format to develop analysis and optimization tools. The memory model of EVM programs has been described in previous work [17,19,26,27]. Mainly, there are three

* This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) projects PID2021-122830OB-C41 and PID2021-122830OA-C44 and by the CM project S2018/TCS-4314 co-funded by EIE Funds of the European Union.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 448–466, 2023.
ttps://doi.org/10.1007/978-3-031-30823-9_23

regions in which data can be stored and accessed: (1) The EVM is a stack-based virtual machine, meaning that most instructions perform computations using the topmost elements in a machine *stack*. This memory region can only hold a limited amount of values, up to 1024 256-bit words. (2) EVM programs store data persistently using a memory region named *storage* that consists of a mapping of 256-bit addresses to 256-bit words and whose contents persist between external function calls. (3) The third memory region is a local volatile memory area that we will refer to as EVM *memory*, and which is the focus of our work. This memory area behaves as a simple word-addressed array of bytes that can be accessed by byte or as a one-word group. The EVM memory can be used to allocate dynamic local data (such as arrays or structs) and also for specific EVM bytecode instructions which have been designed to require some lengthy operands to be stored in local memory. This is the case of the instructions for computing cryptographic hashes, or for passing arguments to and returning data from external function calls. Compilers use the stack and volatile memory regions in different ways. The most-used Solidity compiler `solc` generates EVM code that uses the stack for storing value-type local variables, as well as intermediate values for complex computations and jump addresses, whereas reference-type local variables such as array types and user-defined struct types are located in memory. For instance, when a Solidity function returns a struct variable, the required memory for the struct is allocated and initialized at the beginning of the function execution. However, the allocated memory is not always accessed as we illustrate in the following function (that belongs to the contract in Fig. 1):

```
1 function _ownershipAt(uint256 i) private returns (TokenOwnership memory) {
2   return c.unpackedOwnership(_packedOwnerships[i]);
3 }
```

Although the execution of `_ownershipAt` allocates memory for the return value declared in the function definition, the execution of the function is reserving a different memory space for the actual returned struct obtained from `unpackedOwnership` and, thus, the first reservation and its initialization are needless. The focus of our work is on detecting such needless write memory accesses on the code generated by `solc`. Nevertheless, as the analysis works at EVM level, it could be easily adapted to EVM code generated by any other compiler.

Optimization. Optimization of Ethereum smart contracts is a hot research topic, see e.g. [9,10,12–14,22,24] and their references. This is because the reduction of their costs is relevant for three reasons: (1) *Deployment fees.* When the contract is deployed on the blockchain, the owner pays a fee related to the size in bytes of the bytecode. Hence, a clear optimization criterion is the bytes-size of the program. The Solidity compiler `solc` [6] has as optimization target such bytes-size reduction. (2) *Gas-metered execution.* There is a fee to be paid by each client to execute a transaction in the blockchain. This fee is a fixed amount per transaction plus the cost of executing all bytecode instructions within the function being invoked within the transaction. This cost is measured in "gas" (which is then priced in the corresponding cryptocurrency) and this is why the execution is said to be gas-metered. The EVM specification ([27] and more recent updates)

provides a precise gas consumption for each bytecode instruction in the language. The goal of most EVM bytecode optimization tools [9,10,12–14,22] is to reduce such gas consumption, as this will revert on reducing the price of all transactions on the smart contract. (3) *Enlarging Ethereum's capability.* Due to the huge volume of transactions that are being demanded, there is a huge interest in enlarging the capability of the Ethereum network to increase the number of transactions that can be handled. Optimization of EVM bytecode in general –and of its memory usage in particular– is an important step contributing into this direction.

Challenges and contributions. Optimizing memory usage is considered a challenging problem that requires a precise inference of the memory locations being accessed, and that usually varies according to the memory model of the language being analyzed, and to the compiler that generates the code to be executed. In the case of Ethereum smart contracts generated by the `solc` compiler, the memory model is rather unconventional and its low-level memory usage patterns challenge automated reasoning. On one hand, instead of having an instruction to allocate memory, the allocation is performed by a sequence of instructions that use the value stored at address $0x40$ as the *free memory pointer*, i.e., a pointer to the first memory address available for allocating new memory. In the general case, the memory is structured as a sequence of *slots*: a slot is composed of several consecutive memory locations that are accessed in the bytecode from the same initial memory location plus a corresponding offset. A slot might just hold a data structure created in the smart contract but also, when nested data structures are used, from one slot we can find pointers to other memory slots for the nested components. Finally, there are other type of *transient* slots that hold temporary data and that need to be captured by a precise memory analysis as well. These features pose the main challenges to infer needless write accesses and, to handle them accurately, we make the following main contributions: (1) we present a *slot analysis* to (over-)approximate the slots created along the execution and the program points at which they are allocated; (2) we then introduce a *slot usage analysis* which infers the accesses to the different slots from the bytecode instructions; (3) we finally infer *needless write accesses*, i.e., program points where the memory is written but is never read by any subsequent instruction of the program; and (4) we implement the approach and perform a thorough experimental evaluation on real smart contracts detecting needless write accesses which belong to highly optimizable memory usage patterns generated by `solc`. Finally, it is worth mentioning that the applications of the memory analysis (points 1 and 2) go beyond the detection of needless write accesses: a precise model of the EVM memory is crucial to enhance the accuracy of any posterior analysis (see, e.g., [19] for other concrete applications of a memory analysis).

2 Memory Layout and Motivating Examples

Memory Opcodes. The EVM instruction set contains the usual instructions to access memory: the most basic instructions that operate on memory are `MLOAD`

```
4  struct TokenOwnership {
5    address addr;
6    uint64 startTs;
7    bool burned;
8  }
9
10 contract Running1 {
11   //...
12   function unpackedOwnership
13     (uint256 packed) public
14 s1 s2  returns (TokenOwnership
       memory ownership) {
15     ownership.addr = ...;
16     ownership.startTs = ...;
17     ownership.burned = ...;
18   }
19 }
```

```
17 contract Running2 {
18   Running1 c;
19   mapping(uint256=>uint256) private _packedOwnerships;
20   // ...
21   function _ownershipAt(uint256 i) private
22 s6       returns (TokenOwnership memory) {
23 s7   return c.unpackedOwnership(_packedOwnerships[i]);
24   }
25   function explicitOwnershipOf(uint256 tokenId)
26 s3       public returns (TokenOwnership memory) {
27 s4   TokenOwnership memory ownership;
28 s5   if (...) { return ownership; }
29 s8   ownership = _ownershipAt(tokenId);
30     //...
31 s5   return ownership;
32   }
33 }
```

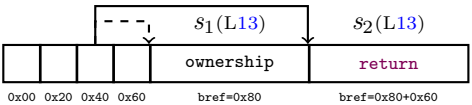
Fig. 1: Excerpt of smart contract ERC721A.

and `MSTORE`, which load and store a 32-byte word from memory, respectively.[3] The `solc` compiler generates code to handle memory with a cumulative model in which memory is allocated along the execution of the program and is never released. In contrast to other bytecode virtual machines, like the *Java Virtual Machine*, the EVM does not have a particular instruction to allocate memory. The allocation is performed by a sequence of instructions that use the value stored at address $0x40$ as the *free memory pointer*, i.e., a pointer to the first memory address available for allocating new memory. In what follows, we use $mem\langle x \rangle$ to refer to the content stored in memory at location x.

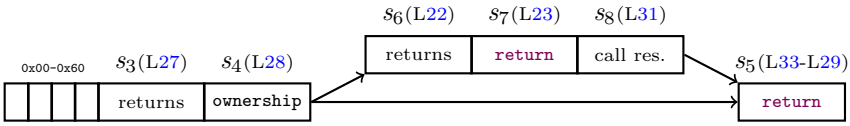
Memory Slots. In the general case, memory is structured as a sequence of *slots*. A slot is composed of consecutive memory locations that are accessed by using its initial memory location, which we call the *base reference* (*baseref* for short) of the slot, plus the corresponding offset needed to access a specific location within the slot. Slots usually store (part of) some data structure created in the Solidity program (e.g., an array or a struct) and whose length can be known.

Example 1 (slots). Fig. 1 shows an excerpt of smart contract ERC721A [2] which contains two different contracts `Running1` and `Running2`. We have omitted non-relevant instructions such as those that appear at lines 15-17 (L15-L17 for short). The contract `Running1` to the left of Fig. 1 contains the public function `unpackedOwnership` that returns a struct of type `TokenOwnership` defined at L4-L7. The contract `Running2`, shown to the right, contains the public function `explicitOwnershipOf` that returns, depending on a non-relevant condition, an empty struct of type `TokenOwnership` (L29) or the `TokenOwnership` received from a call to function `unpackedOwnership` of contract `Running1` (L23), which is done in the private function `_ownershipAt`. The execution of function `unpackedOwnership` in `Running1` allocates two different memory slots at L13: s_1, for the returned variable `ownership`, and s_2, which is used for actually returning from the function the contents of `ownership`:

[3] Although the local memory is byte addressable with instruction `MSTORE8`, to keep the description simpler, we only consider the general case of word-addressable `MSTORE`.



The function `explicitOwnershipOf` in `Running2` makes a more intensive use of the memory which can be seen in this graphical representation:



The execution of this function might create up to six different slots. At L27 and L28, it creates two slots, one for the struct declared in the `returns` part of the function header (s_3) and one for the local variable `ownership` (s_4). Depending on the evaluation of the condition in the `if` sentence, it might create the slots needed to perform the call to `_ownershipAt` and, consequently, the external call to `Running1.unpackedOwnership`. The invocation to the private function involves three slots: one for the struct declared in the `returns` part of `_ownershipAt` in L31 (s_6), one slot to manage the external call data in L23 (s_7), and one slot for storing the results of the private function `_ownershipAt` in L31 (s_8). Finally, a new slot (s_5) is created for returning the results of `explicitOwnershipOf`. This new slot might contain the contents of s_4 or s_8, depending on the `if` evaluation.

When an amount of memory t is to be allocated, the slot reservation is made by reading and incrementing the free memory pointer ($mem\langle 0x40\rangle$) t positions. From this update on, the *base reference* to the slot just allocated is used, and subsequent accesses to the slot are performed by means of this baseref, possibly incremented by an offset.

Example 2 (memory slot reservation). The following excerpt of EVM code allocates a slot of type `TokenOwnership`. The EVM bytecode performs three steps: (i) load the current value of the free memory pointer $mem\langle 0x40\rangle$ that will be used as the *baseref* of the new slot; (ii) compute the new free memory address by adding t to the *baseref*; and (iii), store the new free memory pointer in $mem\langle 0x40\rangle$. Additionally, in the same block of the CFG, the slot reservation is followed by the slot initialization at `0x19A`, `0x1AB` and `0x1B4`.

```
0x175: JUMPDEST
0x176: PUSH1 0x40
0x178: MLOAD     // (i) baseref
       DUP1
       PUSH1 0x60 // Sizeof "t"
       ADD  // (ii) baseref+0x60
0x17D: PUSH1 0x40
0x17F: MSTORE    // (iii)
       ...
0x19A: MSTORE    // baseref+0x00
       ...
0x1AB: MSTORE    // baseref+0x20
       ...
0x1B4: MSTORE    // baseref+0x40
```

Solidity reference type values such as arrays, struct typed variables and strings are stored in memory using this general pattern, with some minor differences. However, there are some cases in which the steps detailed above vary and the size of the slot is not known in advance, and thus the free memory pointer cannot be updated at this point. For instance, when data is returned by an external call, its length is unknown beforehand and hence the free memory pointer is updated only after the memory pointed to is written. In other cases, the free memory is used as a temporary region with a short lifetime, as in the case of parameter

passing to external calls, and the free memory pointer is not updated. These variants of the general schema must be detected by a precise memory analysis. To this end, we consider that a slot is in *transient* state when its baseref has been read from $mem\langle 0x40\rangle$ but the free memory pointer has not been updated, and it is in *permanent* state when the free memory pointer has been pushed forward.

Example 3 (transient slot). Now we focus on the external call in L23 of `Running2`, which performs a `STATICCALL`, reading from the stack (see [27] for details) the memory location of the input arguments and the location where the results of the call will be saved. Interestingly, both locations reuse the same slot (it corresponds to s_7) as it can be seen in the following EVM bytecode from `_ownerShipAt`:

```
    PUSH4 0xb04dd20b // func. selector
    ...
    PUSH1 0x40
0x114: MLOAD   // baseref transient slot
    ...
    DUP2
    MSTORE  // stores func. selector
    PUSH1 0x04
    ADD     // offset of funct. args.
    ...   // copy func. args.
    MSTORE  // stores func. args.
    ...
```

```
    PUSH 0x40
0x132: MLOAD      // slot baseref
    ...
0x139: STATICCALL // external call
    ...
    PUSH1 0x40
0x151: MLOAD      // slot baseref
    RETURNDATASIZE
    ...
    ADD       // baseref + data size
    ...
0x15E: PUSH1 0x40
0x160: MSTORE   // permanent slot
```

The call starts by reading the free memory pointer (at `0x114`) and storing at that address the arguments' data (which include the function selector as first argument). Importantly, the pointer is not pushed forward when the input arguments are written and thus the slot remains in transient state. Once the call at `0x139` is executed, the result is written to memory from the baseref on (overwriting the locations used for the input arguments) and the slot is finally made permanent by reading the free memory pointer again (`0x151`) and updating it (`0x160`) by adding the actual return data size (`RETURNDATASIZE`).

Transient slots are also used when returning data from a public function to an external caller. In that case, the EVM code of the public function halts its execution using a `RETURN` instruction. It reads from the stack the memory location where the length and the data to be returned are located. However, it does not change $mem\langle 0x40\rangle$ because the function code halts its execution at this point, as we can see in the EVM code of `explicitOwnershipOf` (corresponds to slot s_5):

```
    PUSH1 0x40
0x4D: MLOAD   //ret slot baseref
    ...
    MSTORE  // ret.addr (ret+0x00)
    ...
    MSTORE  // ret.startTs (ret+0x20)
    ...
    MSTORE  // ret.burned (ret+0x40)
```

```
    ...
    PUSH1 0x40
0x5A: MLOAD  //ret slot revisit
    DUP1
    SWAP2  //Baseref of ret plus size
    SUB    //Size of ret data
0x5E: SWAP1
0x5F: RETURN //ret returned
```

The baseref for the return slot is read (at `0x4D`) and it is used as a transient slot to write the struct contents to be returned by adding the corresponding offset for each field contained in the struct (instructions on the left column). The code on the left ends with the baseref plus the size of the stored data on top of the stack. After that, the baseref is read again (top of the right column) and the length of the returned data is computed (by subtracting the baseref to the baseref plus the size of the stored data) before calling the `RETURN` instruction.

3 Inference of Needless Write Accesses

This section presents our static inference of needless write accesses. We first provide some background in Sec. 3.1 on the type of control-flow-graph (CFG) and static analysis we rely upon. Then, the analysis is divided into three consecutive steps: (1) the slot analysis, which is introduced in Sec. 3.2, to identify the slots created along the execution and the program points at which they are allocated; (2) the slot usage analysis, presented in Sec. 3.3, which computes the read and write accesses to the different slots identified in the previous step; and (3) the detection of needless write accesses, given in Sec. 3.4, which finds those program points where there is a write access to a slot which has no read access later on.

3.1 Context-Sensitive CFG and Flow-Sensitive Static Analysis

The construction of the CFG of Ethereum smart contracts is a key part of any decompiler and static analysis tool and has been subject of previous research [15, 16, 25]. The more precise the CFG is, the more accurate our analysis results will be. In particular, context-sensitivity [16] on the CFG construction is vital to achieve precise results. Our implementation of context-sensitivity is realized by cloning the blocks which are reached from different contexts.

Example 4 (context-sensitive CFG). The EVM code of `Running2` creates multiple slots for handling structs of type `TokenOwnership`. Interestingly, all these slots are created by means of the same EVM code shown in Ex. 2, which corresponds to the CFG block that starts at program point `0x175`. As this block is reached from different contexts, the context-sensitive CFG contains three clones of this block: `0x175`, which creates s_3 at L27; `0x175_0`, which creates s_4 used at L28; and `0x175_1`, which reserves s_6, created at L22. Block cloning means that program points are cloned as well, and we adopt the same subindex notation to refer to the program points included in the cloned block: e.g. program point `0x178` contains the `MLOAD 0x40` that gets the baseref of the slot reserved at block `0x178`, and `0x178_0` to the same `MLOAD` but at `0x178_0`, etc.

In what follows, we assume that cloning has been made and the memory analysis using the resulting CFG (with clones) is thus context-sensitive as well, without requiring additional extensions. As usual in standard analyses [23], one has to define the notion of *abstract state* which defines the abstract information gathered in the analysis and the *transfer function* which models the analysis output for each possible input. Besides context-sensitivity, the two analyses that we will present in the next two sections are *flow-sensitive*, i.e., they make a flow-sensitive traversal of the CFG of the program using as input for analyzing each block of the CFG the information inferred for its callers. When the analysis reaches a CFG block with new information, we use the operation $\sqcup$ to join the two abstract states, and the operator $\sqsubseteq$ to detect that a fixpoint is reached and, thus, that the analysis terminates. The operations $\sqcup$ and $\sqsubseteq$, the abstract state, and transfer function, will be defined for each particular analysis.

3.2 Slot Analysis

The slot analysis aims at inferring the *abstract slots*, which are an abstraction of all memory allocations that will be made along the program execution. The

slots inferred are *abstract* because over-approximation is made at the level of the program points at which slots are allocated. Therefore, an abstract slot might represent multiple (not necessarily consecutive) real memory slots, e.g., when memory is allocated within a loop. The slot analysis will look for those program points at which the value stored in $mem\langle 0x40\rangle$ is read for reserving memory space. These program points are relevant in the analysis for two reasons: firstly, to obtain the baseref of the memory slot, and, secondly, because from this point on, the memory reservation of the corresponding slot has started and it is pending to become permanent at some subsequent program point. The output of the slot analysis is a set which contains the allocated abstract slots, named $\mathcal{S}_{all}$ in Def. 2 below. Each allocated abstract slot (i.e., each element in $\mathcal{S}_{all}$) is in turn a set of program points, as the same abstract slot might have several program points where $mem\langle 0x40\rangle$ is read before its reservation becomes permanent. In order to obtain $\mathcal{S}_{all}$, the memory analysis makes a flow-sensitive traversal of the (context-sensitive) CFG of the program that keeps at every program point the set of transient slots (i.e. whose baseref has been read but it has not yet made permanent) and applies the transfer function in Def. 1 to each bytecode instruction within the blocks until a fixpoint is reached. An *abstract state* of the analysis is a set $\mathcal{S} \subseteq \ (\mathcal{P}_R)$, where $\mathcal{P}_R$ is the set of all program points at which $mem\langle 0x40\rangle$ is read. The analysis of the program starts with $\mathcal{S} = \{\emptyset\}$ at all program points and takes $\sqcup$ and $\sqsubseteq$ as the set union and inclusion operations. Termination is trivially guaranteed as the number of program points is finite and so is $\ (\mathcal{P}_R)$. In what follows, *Ins* is the set of EVM instructions and, for simplicity, we consider `MLOAD 0x40` and `MSTORE 0x40` as single instructions in *Ins*.

Definition 1 (slot analysis transfer function). *Given a program point pp with an instruction* $I \in Ins$*, an abstract state* $\mathcal{S}$*, and* $\mathcal{K} = \{$*MSTORE 0x40*, *RETURN*, *REVERT*, *STOP*, *SELFDESTRUCT*$\}$*, the* slot analysis transfer function ν *is defined as a mapping* $\nu : Ins \times \ (\mathcal{S}) \mapsto \ (\mathcal{S})$ *computed according to the following table:*

	I	$\nu(\mathbf{I}, \mathcal{S})$
(1)	*MLOAD 0x40*	$\{s \cup \{pp\} \mid s \in \mathcal{S}\}$
(2)	$I \in \mathcal{K}$	$\{\emptyset\}$
(3)	*otherwise*	$\mathcal{S}$

Let us explain intuitively how the above transfer function works. As we have seen in Sec. 2, in an EVM program all memory reservations start by reading $mem\langle 0x40\rangle$ by means of a `MLOAD` instruction preceded by a `PUSH 0x40` instruction (case 1 in Def. 1). In this case, the transfer function adds to all sets in $\mathcal{S}$ the current program point, since this is, in principle, an access to the same slots that were already open at this program point and are not permanent yet. To properly identify the slots, our analysis also searches for those program points at which slots reservations are made permanent (case 2 in Def. 1), i.e., those program points with instructions $I \in \mathcal{K}$. The most frequently used instruction to make a slot reservation *permanent* is a write access to $mem\langle 0x40\rangle$ using `MSTORE`, that pushes forward the free memory pointer such that any subsequent read access to $mem\langle 0x40\rangle$ will allocate a different slot. The rest of instructions in $\mathcal{K}$ finalize the execution in different forms (a normal return, a forced stop, a revert execution, etc.). In all such cases, the slot needs to be considered as a permanent slot so that we can reason later on potential needless write accesses involved in it. The

set $\mathcal{S}$ is empty after these instructions since all transient (abstract) slots are made permanent after them. We use the notation $\mathcal{S}_{pp}$ to refer to the abstract state computed at program point *pp*.

Example 5 (slot analysis). The slot analysis of `Running2` starts with $\mathcal{S}_{pp}$={∅} at all program points. When it reaches the block that starts at `0x175` (see Ex. 2) $\mathcal{S}_{\texttt{0x175}}$ is {∅} and it remains empty until `0x178`, where the baseref of s_3 is read and hence $\mathcal{S}_{\texttt{0x178}}$={{`0x178`}}. This slot is made permanent when the free memory pointer is updated at `0x17F`, thus having $\mathcal{S}_{\texttt{0x17D}}$={{`0x178`}} and $\mathcal{S}_{\texttt{0x17F}}$={∅}. Following the same pattern, s_4 and s_6 are resp. reserved at instructions `0x178_0` and `0x178_1` and closed at `0x17F_0` and `0x17F_1` (at the cloned blocks). On the other hand, the baseref of s_5 is read at two consecutive program points (`0x4D` and `0x5A`) and updated at `0x5F`, and thus, we have $\mathcal{S}_{\texttt{0x4D}}$={{`0x4D`}} and the same until $\mathcal{S}_{\texttt{0x5A}}$={{`0x4D`, `0x5A`}} and again the same until $\mathcal{S}_{\texttt{0x5F}}$={∅}. Finally, after the execution of `STATICCALL` (see Ex. 3) we have three consecutive reads of $mem\langle$0x40$\rangle$ at `0x114`, `0x132` and `0x151` that refer to the same slot s_7, which is made permanent at `0x160`. Therefore, we have $\mathcal{S}_{\texttt{0x151}}$={{`0x114`, `0x132`, `0x151`}} and $\mathcal{S}_{\texttt{0x160}} = \{\emptyset\}$.

Using the transfer function, as mentioned in Sec. 3.1, our analysis makes a flow-sensitive traversal of the (context-sensitive) CFG of the program that uses as input for analyzing each block the information inferred for its callers. When a fixpoint is reached, we have an abstract state for each program point that we use to compute the set of abstract slots allocated in the program, named $\mathcal{S}_{all}$.

Definition 2. *The set of allocated abstract slots* $\mathcal{S}_{all}$ *is defined as* $\mathcal{S}_{all} = \bigcup_{pp \in \mathcal{P}_W} \mathcal{S}_{pp-1}$*, where* $\mathcal{P}_W$ *is the set of all program points pp*:*I where* $I \in \mathcal{K}$.

Example 6 ($\mathcal{S}_{all}$ computation). With the values of $\mathcal{S}_{\texttt{0x17F-1}}$, $\mathcal{S}_{\texttt{0x17F\_0-1}}$, $\mathcal{S}_{\texttt{0x17F\_1-1}}$, $\mathcal{S}_{\texttt{0x160-1}}$ and $\mathcal{S}_{\texttt{0x5F-1}}$ from Ex. 5, at the end of the slot analysis of `Running2`, we have: $\mathcal{S}_{all}$={$\underbrace{\{\texttt{0x178}\}}_{s_3}, \underbrace{\{\texttt{0x178\_0}\}}_{s_4}, \underbrace{\{\texttt{0x178\_1}\}}_{s_6}, \underbrace{\{\texttt{0x114}, \texttt{0x132}, \texttt{0x151}\}}_{s_7}, \underbrace{\{\texttt{0x5A}, \texttt{0x4D}\}}_{s_5}, \ldots$}.
Note that, the cloning of block `0x175` allows our analysis to detect three different slots, s_3, s_4 and s_6, for the same program point, `0x178`, in the original EVM code.

The next example shows the behavior of the analysis when the program contains loops, and an abstraction is needed for approximating the slots.

Example 7 (loops). Fig. 2 shows the contract `Running3` that includes the function `explicitOwnershipsOf` from the smart contract at [2] (made through a `STATICCALL`). This function receives an array of token identifiers as argument and returns an array of `TokenOwnership` structs that is populated invoking the function `explicitOwnershipOf` from `Running2` inside a loop. The slots identified by the analysis for contract `Running3` shown in Fig. 2 are: s_9, which is created for making a copy of parameter `tokenIds` to memory; s_{10}, which creates the local array `ownerships` (L44) that contains the array length and pointers to the structs identified initially by s_{11} (and later on by s_{13}); s_{12} for `STATICCALL` input arguments and return data (L46); s_{13} which abstracts the structs for storing the `STATICCALL` output results (L46); and s_{14}, which includes the length of ownership

```
37 contract Running3 {
38   Running2 c;
39   //...
40 s9 function explicitOwnershipsOf(uint256[] memory tokenIds)
41     public view returns (TokenOwnership[] memory) {
42     unchecked {
43       uint256 tokenIdsLength = tokenIds.length;
44 s10s11 TokenOwnership[] memory ownerships = new TokenOwnership[](tokenIdsLength);
45       for (uint256 i; i != tokenIdsLength; ++i) {
46 s12s13       ownerships[i] = c.explicitOwnershipOf(tokenIds[i]);
47       }
48 s14   return ownerships;
49     }
50   }
51 }
```

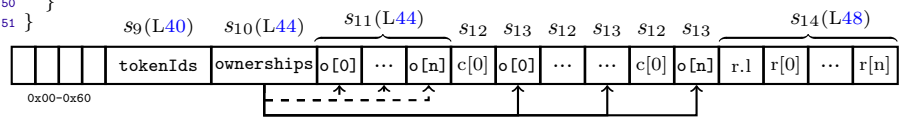


Fig. 2: Solidity code of contract `Caller`.

and a copy of s_{13} for returning the results (L48). The important point is that, the local array declaration at L44 produces a loop to allocate as many structs as elements are contained in the array. For this reason, s_{11} is an abstract slot that represents all `TokenOwnership`'s initially added to the array. Similarly, s_{12} and s_{13} are created inside the `for` loop, and each abstract slot represents as many concrete slots as iterations are performed by the loop. Note that, each iteration of the loop creates one instance of s_{12} for getting the results from the call, and it is copied later to s_{13} and pointed by `ownerships` (s_{10}).

As notation, we will use a unique numeric identifier (1, 2, ...) to refer to each abstract slot (represented in $\mathcal{S}_{all}$ as a set) and retrieve it by means of function $get\_id(a), a \in \mathcal{S}_{all}$. We use $\mathcal{A}$ to refer to the set of all such identifiers in the program. Also, given a program point pp with an instruction `MLOAD 0x40`, we define the function $get\_slots(pp)$ to retrieve the identifiers of the elements of $\mathcal{S}_{all}$ that might be referenced at pp as follows: $get\_slots(pp) = \{id \mid a \in \mathcal{S}_{all} \wedge pp \in a \wedge id = get\_id(a)\}$.

3.3 Slot Access Analysis

While Sec. 3.2 looked for allocations, the next step of the analysis is the inference of the program points at which the inferred abstract slots might be accessed. To do so, our slot access analysis needs to propagate the references to the abstract slots that are saved at the different positions of the execution stack. Importantly, we keep track, not only of the stack positions, but also, in order to abstract complex data structures stored in memory (e.g., arrays of structs), we need to keep track of the abstract slots that could be saved at memory locations. As seen in Ex. 7, a memory location within a slot might contain a pointer to another memory location of another slot, as it happens when nested data structures are used. Thus, an abstract state is a mapping at which we store the potential slots saved at stack positions or at memory locations within other slots.

Definition 3 (memory analysis abstract state). *A memory analysis abstract state is a mapping π of the form $\mathcal{T} \cup \mathcal{A} \mapsto \quad (\mathcal{A})$.*

$\mathcal{T}$ is the set containing all stack positions, which we represent by natural numbers from 0 (bottom of the stack) on, and $\mathcal{A}$ is the set of abstract slots identifiers computed in Sec. 3.2. We refer to the set of all memory analysis abstract states as AS. Note that, for each entry, we keep a set of potential slots for each stack position because a block might be reached from several blocks with different execution stacks, e.g., in loops or *if-then-else* structures. In what follows, we assume that, given a value k, the map π returns the empty set when $k \notin dom(\pi)$. The inference is performed by a flow-sensitive analysis (as described in Sec. 3.1) that keeps track of the information about the abstract slots used at any program point by means of the following transfer function.

Definition 4 (memory analysis transfer function). *Given an instruction I with n input operands at program point pp and an abstract state π, the* memory analysis transfer function τ *is defined as a mapping τ:$Ins \times AS \mapsto AS$ of the form:*

	I	$\tau(I,\pi)$
(1)	`MLOAD 0x40`	$\pi[t \mapsto get\_slots(pp)]$
(2)	`MLOAD`	$\pi[t \mapsto \{m \mid s \in \pi(t) \wedge m \in \pi(s)\}]$
(3)	`MSTORE`	$\pi[s \mapsto \pi(s) \cup \pi(t-1)] \backslash \{t, t-1\}\ \forall s \in \pi(t)$

	I	$\tau(I,\pi)$
(4)	`SWAPi`	$\pi[t \mapsto \pi(t-i), t-i \mapsto \pi(t)]$
(5)	`DUPi`	$\pi[t+1 \mapsto \pi(t-i+1)]$
(6)	*otherwise*	$\pi \backslash x \qquad t-n < x \leq t$

t=$top(pp)$ is the numerical position of the top of the stack before executing I.

Let us explain the above definition. The transfer function distinguishes between two different types of `MLOAD`: (1) accesses to location $mem\langle 0x40\rangle$, which return the baseref of the slots that might be used, taking them from the previous analysis through $get\_slots(p)$; and (2) other `MLOAD` instructions, which could potentially return slot basereﬁs from memory locations. Therefore, we have to consider two possibilities: if we are reading a memory location which reads a generic value (e.g. a number) then $\pi(t) = \emptyset$; if we are reading a memory location that might store an abstract slot, then $\pi(t)$ contains all abstract slots that might be stored at that memory location. Regarding (3), `MSTORE` has two operands: the operand at t is the memory address that will be modified by `MSTORE`, and the operand at $t-1$ is the value to be stored in that address. For each element s in $\pi(t)$, the analysis adds the abstract slots that are in $\pi(t{-}1)$. Other instructions that are also treated by the analysis are `SWAP*` and `DUP*` shown in (4-5), that exchange or copy the elements of the stack that take part in the operation. Finally, all other operations delete the elements of the stack that are no longer used based on the number of elements taken and written to the stack (case 6).

Example 8 (transfer). Now we focus on the analysis of block `0x175`, shown in Fig. 3. As we have already explained, this block is responsible for creating the memory needed to work with several structs of type `TokenOwnership` and it is thus cloned in the CFG. In particular, we focus on the clone `0x175_1`. The analysis of the block starts with a stack of size 7 and includes at positions 3 and 4, the abstract slots s_3 and s_4, which were created at L26 and L27 of Fig. 1. At `0x178_1`, $mem\langle 0x40\rangle$ is read, and, by means of get_slots(`0x178_1`) and, considering that top(`0x178_1`)=8, we add to π a new entry $8 \mapsto s_6$. At `0x179_1`, `0x180_1`, `0x1AA_1`, `0x1B3_1` the transfer function duplicates a slot identifier stored in the stack. `MSTORE` and `POP` instructions of the example remove a slot identifier from the stack.

PP	Instr	π
0x175_1	JUMPDEST	$\{3\mapsto s_3, 4\mapsto s_4\}$
0x176_1	PUSH1 0x40	$\{3\mapsto s_3, 4\mapsto s_4\}$
0x178_1	MLOAD	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6\}$
0x179_1	DUP1	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
0x17A_1	PUSH1 0x60	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
0x17C_1	ADD	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
0x17D_1	PUSH1 0x40	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
0x17F_1	MSTORE	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6\}$
0x180_1	DUP1	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
...		
0x198_1	AND	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
0x199_1	DUP2	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6, 11\mapsto s_6\}$

PP	Instr	π
0x19A_1	MSTORE	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
...		
0x1A9_1	AND	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
0x1AA_1	DUP2	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6, 11\mapsto s_6\}$
0x1AB_1	MSTORE	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
...		
0x1B2_1	ISZERO	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
0x1B3_1	DUP2	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6, 11\mapsto s_6\}$
0x1B4_1	MSTORE	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$
0x1B5_1	POP	$\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6\}$
0x1B6_1	SWAP1	$\{3\mapsto s_3, 4\mapsto s_4, 7\mapsto s_6\}$
0x1B7_1	JUMP	$\{3\mapsto s_3, 4\mapsto s_4, 7\mapsto s_6\}$

Fig. 3: Block of the CFG that reserves memory slot for struct

As it is flow-sensitive, the analysis of each block of the CFG takes as input the join $\sqcup$ of the abstract states computed with the transfer function for the blocks that jump to it, and keeps applying the memory analysis transfer function until a fixpoint is reached. The operation $A \sqcup B$ is the result of joining, by means of operation $\cup$, all entries from maps A and B. Operation $\sqsubseteq$ is defined as expected, $A \sqsubseteq B$, when B includes entries that are not in $dom(A)$ or when we have an entry $v \in dom(A) \cap dom(B)$ such that $A(v) \subseteq B(v)$. Again, termination of the computation is guaranteed because the domain is finite.

Example 9 (joining abstract states). The EVM code of `explicitOwnershipOf` of Fig. 1 uses s_5 in both `return` sentences at L29 and L33 (see Ex. 1). This EVM code has a single return block which is reachable from two different paths from the `if` statement, and which come with different abstract states: (1) the path that corresponds to L29 comes with π=$\{3 \mapsto s_8\}$, and the other path (L33) with π=$\{3 \mapsto s_4\}$. Our analysis joins both abstract states resulting in π=$\{3 \mapsto \{s_4, s_8\}\}$. Because of this join, we get that the `RETURN` instruction that comes from lines L29 and L33 might return the content of the slots s_4 or s_8.

When the fixpoint is reached, the analysis has computed an abstract state for each program point pp, denoted by π_{pp} in what follows.

Example 10 (complex data structures). The analysis of the code at Fig. 2 shows how it deals with data structures that might contain pointers to other structures, e.g. `ownerships`. The abstract slot that represents variable `ownerships` is s_{10}, which is written, by means of `MSTORE` at two program points, say pp_1 and pp_2 which, resp., come from L44 and L46 of the Solidity code. The input abstract state that reaches pp_1 is $\{2 \mapsto s_9, 6 \mapsto s_{10}, 8 \mapsto s_{10}, 9 \mapsto s_{11}, 10 \mapsto s_{10}\}$, and the transfer function of `MSTORE` leaves the abstract state as $\pi_{pp_1} = \{2 \mapsto s_9, 6 \mapsto s_{10}, 8 \mapsto s_{10}, s_{10} \mapsto s_{11}\}$. At this point, we can see that variable `ownerships` is initialized with empty structs and, to represent it, our analysis includes in π the entry $s_{10} \mapsto s_{11}$ as it is described in instruction `MSTORE` of the transfer function at Def. 4. The second write to s_{10} is performed by another `MSTORE` instruction at pp_2. The input abstract state for pp_2 is $\{2 \mapsto s_9, 5 \mapsto s_{10}, 7 \mapsto s_{13}, 8 \mapsto s_{13}, 9 \mapsto s_{10}, s_{10} \mapsto s_{11}\}$, and thus we get $\pi_{pp_2} = \{2 \mapsto s_9, 5 \mapsto s_{10}, 7 \mapsto s_{13}, s_{10} \mapsto \{s_{11}, s_{13}\}\}$. Interestingly, at pp_2, we detect that s_{11} might also store the structs returned by the call to `c.explicitOwnershipOf(tokenIds[i])`, identified by s_{13}, which is added to

$s_{10} \mapsto \{s_{11}, s_{13}\}$. Finally, s_{10} is read at the end of the method, returning the set $\{s_{11}, s_{13}\}$, to copy the content of `ownerships` to s_{14}, the slot used in the return.

3.4 Inference of Needless Write Memory Accesses

With the results of the previous analysis, we can compute the maps $\mathcal{R}$ and $\mathcal{W}$, which are of the form $pp \mapsto \ (\mathcal{A})$ and capture the slots that might be read or written, resp., at the different program points. To do so, as multiple EVM instructions, e.g. `RETURN`, `CALL`, `LOG`, `CREATE`, ..., might read, or write, memory locations taking the concrete location from the stack, we define functions $mr(I)$ and $mw(I)$ that, given an EVM instruction I, return the position in the stack of the address to be read and written by I, resp. If the instruction does not read/write any memory position, function $mr(I) = \bot / mw(I) = \bot$. For example, $mr(\texttt{MLOAD}) = 0$ as it reads the top of the stack and $mw(\texttt{MLOAD}) = \bot$, or $mr(\texttt{STATICCALL}) = 2$ and $mw(\texttt{STATICCALL}) = 4$. Now, we define the read/write maps $\mathcal{R}/\mathcal{W}$:

Definition 5 (memory read/write accesses map). *Given an EVM program P, such that $pp \equiv I \in P$ and being t=top(pp), we define maps $\mathcal{R}$ and $\mathcal{W}$ as follows:*

$$\mathcal{R}(pp) = \begin{cases} \emptyset & mr(I) = \bot \\ \pi_{pp-1}(t - mr(I)) & otherwise \end{cases} \qquad \mathcal{W}(pp) = \begin{cases} \emptyset & mw(I) = \bot \\ \pi_{pp-1}(t - mw(I)) & otherwise \end{cases}$$

Example 11 ($\mathcal{R}/\mathcal{W}$ maps). Let us illustrate the computation of $\mathcal{R}(\texttt{0x139})$ and $\mathcal{W}(\texttt{0x139})$, which contains the `STATICCALL` of `Running2`. With the analysis information obtained from the analysis we have that $top(\texttt{0x139}) = 16$ and $\pi_{\texttt{0x138}} = \{3 \mapsto s_3, 4 \mapsto s_4, 7 \mapsto s_6, 10 \mapsto s_7, 12 \mapsto s_7, 14 \mapsto s_7\}$, thus we get $\mathcal{R}(\texttt{0x139}) = \{s_7\}$ and $\mathcal{W}(\texttt{0x139}) = \{s_7\}$, i.e., the slot used for managing the input and the output of the external call. Analogously, we get that $\mathcal{R}(\texttt{0x178}) = \{s_3\}$ and $\mathcal{W}(\texttt{0x178}) = \emptyset$.

The last step of our analysis consists in searching for write accesses to slots which will never be read later. To do so, we use the information computed in $\mathcal{R}$ and $\mathcal{W}$. Given the CFG of the program and two program points p and $p2$, we define function $reachable(p, p2)$, which returns *true* when there exists a path in the CFG from p to $p2$. We define the set *write leaks* $\mathcal{N}$ as follows:

Definition 6. *Given an EVM program and its $\mathcal{W}$ and $\mathcal{R}$, we define $\mathcal{N}$ as*
$\mathcal{N} = \{pw{:}s \mid pw \in P \wedge s \in \mathcal{W}(pw) \wedge \neg exists\_read(pw, s)\}$
where $exists\_read(pw, s) \equiv \exists\ pr \in dom(\mathcal{R}) \mid s \in \mathcal{R}(pr) \wedge reachable(pw, pr)$.

Intuitively, the set $\mathcal{N}$ contains those write accesses, taken from $\mathcal{W}$, that are never read by subsequent blocks in the CFG. As both function *reachable* and the sets $\mathcal{W}$ and $\mathcal{R}$ are over-approximations, the computation of $\mathcal{N}$ provides us those write accesses that can be safely removed, as the next example shows.

Example 12. Our analysis detects that at program points `0x19A`, `0x1AB` and `0x1B4` there are `MSTORE` operations that are never read in the subsequent blocks of the CFG. Such operations correspond to the memory initialization of s_3, which is performed at L27 of the code of Fig. 1 (see Ex. 2). Given that these write accesses are the only use of the slot, the whole reservation can be safely removed.

Moreover, the analysis detects that program points `0x19A_1`, `0x1AB_1` and `0x1B4_1`, which correspond to the reservation of s_6 performed at L22, are detected as needless. In essence, it means that s_3 and s_6 are allocated and initialized but are never used in the program. Note that, all these program points belong to two blocks cloned: (`0x175` and `0x175_1`). However, the three `MSTORE` operations of the other clone of the same block (`0x175_0`), which correspond to the allocation at L28 are not identified as non-read, as they might be used in the return of the function. For this, the precision of the context-sensitive CFG is necessary to identify these `MSTORE` operations as needless. As a result we cannot eliminate the block because it is needed in one of the clones, but still we can achieve an important optimization on the EVM code by removing the unconditional jumps to this block in the other two cases that would avoid completely the execution of all these instructions (and their corresponding gas consumption [27]).

The soundness of slots and slots access analyses states that, for each concrete slot, there exists an abstract slot in $\mathcal{S}_{all}$ that represents it and, that any access to memory is approximated by an inferred abstract slot. Technical details can be found in an extended report [8].

4 Experimental Evaluation

This section reports on the results of the experimental evaluation of our approach, as described in Sec. 3. All components of the analysis are implemented in Python, are open-source, and can be downloaded from github where detailed instructions for its installation and usage are provided[4]. We use external components to build the CFGs (as this is not a contribution of our work). Our analysis tool accepts smart contracts written in versions of Solidity up to 0.8.17 and bytecode for the Ethereum Virtual Machine v1.10.25[5]. The experiments have been performed on an AMD Ryzen Threadripper PRO 3995WX 64-cores and 512 GB of memory, running Debian 5.10.70. In order to experimentally evaluate the analysis, we pulled from etherscan.io [5] the Ethereum contracts bound to the last 5,000 open-source verified addresses whose source code was available on July 14, 2022. From those addresses, the code of 2.18% of them raises a compilation error from solc. For the code bound to the 4,891 remaining addresses, the generation of the CFG (which is not a contribution of this work) timeouts after 120s on 626 of them. Removing such failing cases, we have finally analyzed 19,199 smart contracts, as each address and each Solidity file may contain several contracts in it. Note that 84.86% of the contracts are compiled with the solc version 0.8, presumably with the most advanced compilation techniques. The whole dataset used will be found at the above github link.

In order to be in a worst-case scenario for us, we run the memory analysis after executing the `solc` optimizer, i.e, we analyze bytecode whose memory usage may have been optimized already by the optimizer available in `solc`. This will allow us also to see if we can achieve further optimization with our

[4] https://github.com/costa-group/EthIR/tree/memory_optimizer/ethir
[5] The latest versions released up to Oct 2022.

approach. Unfortunately, we have not been able to apply our tool after running the super-optimizer GASOL [9], because it does not generate the optimized bytecode but rather it only reports on the gas and/or size gains for each of the blocks. Nevertheless, a detailed comparison of the techniques that GASOL applies and ours is given in Sec. 5, where we justify that GASOL will not find any of our needless accesses. From the 19,199 analyzed contracts, the analysis infers 679,517 abstract memory slots and detects 6,242 needless write memory accesses in 12,803s. These needless accesses occur within the code bound to 780 different addresses, i.e., 15.95% of the analyzed ones.

We have computed the number of needless accesses identified by our analysis grouped by function and the number of different contracts that contain these functions. Some of them such as transferFrom(1736 accesses in 439 contracts), transfer (1745 aacesses in 441 contracts), reflectionFromToken(105 accesses in 6 contracts) or withdraw(54 accesses in 32 contracts) are functions widely used in the implementation of contracts based on ERC tokens. A manual inspection of the 10 most common public functions with the needless accesses inferred has revealed two different sources for them: some of the needles accesses are due to inefficient programming practices, while others are generated by the compiler and could be improved. As regards compiler inefficiencies, we detected bytecode that allocates memory slots that are inaccessible and cannot be used because the baseref to access them is not maintained in the stack. For example, when a struct is returned by a function, it always allocates memory for this data. However, if the return variable is not named in the header of the function, the compiler allocates memory for this data although it will never be accessed. If programmers are aware of this behavior they can avoid such generation of useless memory but, even better, this memory usage patterns can be changed in the compiler. For instance, it is reflected in L22 and L27 in Fig. 1, where the functions do not name the return variable. Hence, the compiler allocates memory for these *anonymous* data structures which are never used. Similarly, there are various situations involving external calls in which the compiler creates memory that is never used. When there is an external call that does not retrieve any result, the compiler creates two memory slots, one for retrieving the result from the call, and another one for copying a potential result to a memory variable that is never used. Finally, the compiler also creates memory that is never used for low-level plain calls for currency transfer. Even though the contract code does not use the second result returned by the low-level call, the compiler generates code for retrieving it. All these potential optimizations have been detected by means of our inference of needless write accesses and will be communicated to the `solc` developers.

5 Conclusions and Related Work

We have proposed a novel memory analysis for Ethereum smart contracts and have applied it to infer needless write memory accesses. The application of our implementation over more than 19,000 real smart contracts has detected some compilation patterns that introduce needless write accesses and that can be easily

changed in the compiler to generate more efficient code. Let us discuss related work along two directions: (1) memory analysis and (2) memory optimization. Regarding (1), we can find advanced points-to analysis developed for Java-like languages [7, 11, 18, 20]. Focusing on EVM, the static modeling of the EVM memory in [16] has some similarities with the memory analysis presented in Secs. 3.2 and 3.3, since in both cases we are seeking to model the memory although with different applications in mind. There are differences on one hand on the type of static analysis used in both cases: [16] is based on a Datalog analysis while we have defined a standard transfer function which is used within a flow-sensitive analysis. More importantly, there are differences on the precision of both analyses. We can accurately model the memory allocated by nested data structures in which the memory contains pointers to other memory slots, while [16] does not capture such type of accesses. This is fundamental to perform memory optimization since, as shown in the running examples of the paper, it allows detecting needless write accesses that otherwise would be missed. Finally, the application of the memory analysis to optimization is not studied in [16], while it is the main focus of our work.

As regards (2), optimizing memory usage is a challenging research problem that requires to precisely infer the memory positions that are being accessed. Such positions sometimes are statically known (e.g., when accessing the EVM free memory pointer) but, as we have seen, often a precise and complex inference is required to figure out the slot being accessed at each memory access bytecode. Recent work within the super-optimizer GASOL [9] is able to perform some memory optimizations at the level of each block of the CFG (i.e., intra-block). of There are three fundamental differences between our work and GASOL: First, GASOL can only apply the optimizations when the memory locations being addressed refer to the same constant direction. In other words, there is no real memory analysis (namely Secs. 3.2 and 3.3). Second, the optimizations are applied only at an intra-block level and hence many optimization opportunities are missed. These two points make a fundamental difference with our approach, since detected optimizable patterns (see Sec. 4) require inter-block analysis and a precise slot access analysis, and hence cannot be detected by GASOL.

Finally, as mentioned in Sec. 1, in addition to dynamic memory, smart contracts also use a persistent memory called storage. Regarding the application of our approach to infer needless accesses in storage, there are two main points. First, there is no need to develop a static analysis to detect the slots in storage, as they are statically known (hence our inference in Sec. 3.2 and 3.3 is not needed), i.e., one can easily know the read and write sets of Def. 6. Thus, the read and write sets of our analysis can be easily defined for storage. The second point is that, as storage is persistent memory, a write storage access is not removable even if there is no further read access within the smart contract, as it needs to be stored for a future transaction. The removable write storage accesses are only those that are rewritten and not read in-between the two write accesses. Including this in our implementation is straightforward. However, this situation is rather unusual, and we believe that very few cases would be found and hence little optimization can be achieved.

References

1. Bamboo. https://github.com/pirapira/bamboo.
2. ERC721A. https://etherscan.io/address/0xfcd5c0ef90715dc052dad6de08efda758aa09f60#code.
3. Serpent. https://github.com/ethereum/wiki/wiki/Serpent.
4. Vyper. https://github.com/ethereum/vyper.
5. Etherscan. https://etherscan.io, 2018.
6. Solidity documentation, 2021. https://docs.soliditylang.org/en/latest/index.html.
7. Elvira Albert, Puri Arenas, Jesús Correas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla an d Guillermo Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.
8. Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Inferring needless write memory accesses on ethereum bytecode (extended version), 2023. arXiv:2301.04757 [cs.PL].
9. Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, and Albert Rubio. A Max-SMT Superoptimizer for EVM handling Memory and Storage. In Dana Fisman and Grigore Rosu, editors, *28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022. Proceedings*, volume 13243 of *Lecture Notes in Computer Science*, pages 201–219. Springer, 2022.
10. T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski. Characterizing Efficiency Optimizations in Solidity Smart Contracts. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 281–290, 2020.
11. Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Relevant context inference. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 133–146. ACM, 1999.
12. Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, PP(99):1–14, 03 2020.
13. Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 81–84, 2018.
14. Bo Gao, Siyuan Shen, Ling Shi, Jiaying Li, Jun Sun, and Lei Bu. Verification assisted gas reduction for smart contracts. In *28th Asia-Pacific Software Engineering Conference, APSEC 2021, Taipei, Taiwan, December 6-9, 2021*, pages 264–274. IEEE, 2021.
15. Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1176–1186. IEEE / ACM, 2019.
16. Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: Advanced decompilation of ethereum smart contracts. *Proc. ACM Program. Lang.*, 6(OOPSLA):77:1–77:27, 2022.

17. Ákos Hajdu and Dejan Jovanovic. Smt-friendly formalization of the solidity memory model. In Peter Müller, editor, *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 224–250. Springer, 2020.
18. George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 423–434. ACM, 2013.
19. Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. Precise static modeling of ethereum "memory". *Proc. ACM Program. Lang.*, 4(OOPSLA):190:1–190:26, 2020.
20. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering Methodology*, 14:1–41, 2005.
21. Mayukh Mukhopadhyay. *Ethereum Smart Contract Development*. Packt publishing, 2018.
22. Julian Nagele and Maria A Schett. Blockchain superoptimizer. In *Preproceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)*, 2019.
23. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
24. Maria A. Schett and Julian Nagele. Populating the Peephole Optimizer of a Smart Contract Compiler. In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *OpenAccess Series in Informatics (OASIcs)*, pages 3:1–3:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.
25. Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. eThor: Practical and provably sound static analysis of ethereum smart contracts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, USA, November 9-13, 2020*, pages 621–640. ACM, 2020.
26. Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical Security Analysis of Smart Contracts. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 67–82. ACM, 2018.
27. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2019.

Markov Chains/Stochastic Control

A Practitioner's Guide to MDP Model Checking Algorithms*

Arnd Hartmanns[1], Sebastian Junges[2], Tim Quatmann[3], and Maximilian Weininger[4](✉)

[1] University of Twente, Enschede, The Netherlands a.hartmanns@utwente.nl
[2] Radboud University, Nijmegen, The Netherlands sebastian.junges@ru.nl
[3] RWTH Aachen University, Aachen, Germany tim.quatmann@cs.rwth-aachen.de
[4] Technical University of Munich, Munich, Germany maxi.weininger@tum.de

Abstract. Model checking undiscounted reachability and expected-reward properties on Markov decision processes (MDPs) is key for the verification of systems that act under uncertainty. Popular algorithms are policy iteration and variants of value iteration; in tool competitions, most participants rely on the latter. These algorithms generally need worst-case exponential time. However, the problem can equally be formulated as a linear program, solvable in polynomial time. In this paper, we give a detailed overview of today's state-of-the-art algorithms for MDP model checking with a focus on performance and correctness. We highlight their fundamental differences, and describe various optimizations and implementation variants. We experimentally compare floating-point and exact-arithmetic implementations of all algorithms on three benchmark sets using two probabilistic model checkers. Our results show that (optimistic) value iteration is a sensible default, but other algorithms are preferable in specific settings. This paper thereby provides a guide for MDP verification practitioners—tool builders and users alike.

1 Introduction

The verification of MDPs is crucial for the design and evaluation of cyber-physical systems with sensor noise, biological and chemical processes, network protocols, and many other complex systems. MDPs are the standard model for sequential decision making under uncertainty and thus at the heart of reinforcement learning. Many dependability evaluation and safety assurance approaches rely in some form on the verification of MDPs with respect to temporal logic properties. Probabilistic model checking [4,5] provides powerful tools to support this task.

The essential MDP model checking queries are for the *worst-case probability that something bad happens* (reachability) and the *expected resource consumption until task completion* (expected rewards). These are *indefinite (undiscounted)*

* This research was funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233 (MISSION), and by NWO VENI grant no. 639.021.754.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 469–488, 2023.
https://doi.org/10.1007/978-3-031-30823-9_24

horizon queries: They ask about the probability or expectation of a random variable up until an event—which forms the horizon—but are themselves unbounded. Many more complex properties internally reduce to solving either reachability or expected rewards. For example, if the description of *something bad* is in linear temporal logic (LTL), then a product construction with a suitable automaton reduces the LTL query to reachability [6]. This paper sets out to determine the practically best algorithms to solve indefinite horizon reachability probabilities and expected rewards; our methodology is an empirical evaluation.

MDP analysis is well studied in many fields and has lead to three main types of algorithms: *value iteration* (VI), *policy iteration* (PI), and *linear programming* (LP) [55]. While indefinite horizon queries are natural in a verification context, they differ from the standard problem of e.g. operations research, planning, and reinforcement learning. In those fields, the primary concern is to *compute a policy* that (often approximately) optimizes the *discounted* expected reward over an infinite horizon where rewards accumulated in the future are weighted by a discount factor < 1 that exponentially prefers values accumulated earlier.

The lack of discounting in verification has vast implications. The *Bellman operation*, essentially describing a one-step backward update on expected rewards, is a contraction with discounting, but not a contraction without. This leads to significantly more complex termination criteria for VI-based verification approaches [34]. Indeed, VI runs in polynomial time for every fixed discount factor [49], and similar results are known for PI as well as LP solving with the simplex algorithm [60]. In contrast, VI [9] and PI [20] are known to have exponential worst-case behaviour in the undiscounted case.

So, *what is the best algorithm for model checking MDPs?* A polynomial-time algorithm exists using an LP formulation and barrier methods for its solution [12]. LP-based approaches (and their extension to MILPs) are also prominent for multi-objective model checking [21], in counterexample generation [23], and for the analysis of parametric Markov chains [16]. However, folklore tells us that iterative methods, in particular VI, are better for solving MDPs. Indeed, variations of VI are the default choice of all model checkers participating in the QComp competition [14]. This uniformity may be misleading. Indeed, for some stochastic game algorithms, using LP to solve the underlying MDPs may be preferential [3, Appendix E.4]. An application in runtime assurance preferred PI for numerical stability [45, Sect. 6]. A toy example from [34] is a famous challenge for VI-based methods. Despite the prominence of LP, the ease of encoding MDPs, and the availability of powerful off-the-shelf LP solvers, many tools did (until very recently) not include MDP model checking via LP solvers.

With this paper, we reconsider the PI and LP algorithms to investigate whether probabilistic model checking focused on the wrong family of algorithms. We report the results of an extensive empirical study with two independent implementations in the model checkers Storm [42] and mcsta [37]. We find that, in terms of performance and scalability, optimistic value iteration [40] is a solid choice on the standard benchmark collection (which goes beyond competition benchmarks) but can be beat quite considerably on challenging cases. We also

emphasize the question of precision and soundness. Numerical algorithms, in particular ones that converge *in the limit*, are prone to delivering wrong results. For VI, the recognition of this problem has led to a series of improvements over the last decade [8,34,40,19,54,56]. We show that PI faces a similar problem. When using floating-point arithmetic, additional issues may arise [36,59]. Our use of various LP solvers exhibits concerning results for a variety of benchmarks. We therefore also include results for *exact* computation using rational arithmetic.

Limitations of this study. A thorough experimental study of algorithms requires a carefully scoped evaluation. We work with flat representations of MDPs that fit completely into memory (i.e. we ignore the state space exploration process and symbolic methods). We selected algorithms that are tailored to converge to *the* optimal value. We also exclude approaches that incrementally build and solve (partial or abstract) MDPs using simulation or model checking results to guide exploration: they are an orthogonal improvement and would equally profit from faster algorithms to solve the partial MDPs. Moreover, this study is on algorithms, not on their implementations. To reduce the impact of potential implementation flaws, we use two independent tools where possible. Our experiments ran on a single type of machine—we do not study the effect of different hardware.

Contributions. This paper contributes a thorough overview on how to model-check indefinite horizon properties on MDPs, making MDP model checking more accessible, but also pushing the state-of-the-art by clarifying open questions. Our study is built upon a thorough empirical evaluation using two independent code bases, sources benchmarks from the standard benchmark suite and recent publications, compares 10 LP solvers, and studies the influence of various prominent preprocessing techniques. The paper provides new insights and reviews folklore statements: Particular highlights are a new simple but challenging MDP family that leads to wrong results on all floating-point LP solvers (Section 2.3), a negative result regarding the soundness of PI with epsilon-precise policy evaluators (Section 4), and an evaluation on numerically challenging benchmarks that shows the limitations of value iteration in a practical setting (Section 5.3).

2 Background

We recall MDPs with reachability and reward objectives, describe solution algorithms and their guarantees, and address commonly used optimizations.

2.1 Markov Decision Processes

Let $\mathsf{D}_X \coloneqq \{\, \mathsf{d} \colon X \to [0,1] \mid \sum_{x \in X} \mathsf{d}(x) = 1 \,\}$ be the set of distributions over X. A Markov decision process (MDP) [55] is a tuple $\mathcal{M} = (\mathsf{S}, \mathsf{A}, \delta)$ with finite sets of states S and actions A, and a partially defined transition function $\delta \colon \mathsf{S} \times \mathsf{A} \rightharpoonup \mathsf{D}_\mathsf{S}$ such that $\mathsf{A}(s) \coloneqq \{\, a \mid (s,a) \in \mathit{domain}(\delta) \,\} \neq \emptyset$ for all $s \in \mathsf{S}$. $\mathsf{A}(s)$ is the set of enabled actions at state s. δ maps enabled state-action pairs to distributions over successor states. A Markov chain (MC) is an MDP with $|\mathsf{A}(s)| = 1$ for all s. The *semantics* of an MDP are defined in the usual way, see, e.g. [6, Chapter 10]. A

(memoryless deterministic) policy—a.k.a. strategy or scheduler—is a function $\pi\colon \mathsf{S} \to \mathsf{A}$ that, intuitively, given the current state s prescribes what action $a \in \mathsf{A}(s)$ to play. Applying a policy π to an MDP induces an MC $\mathcal{M}^\pi$. A path in this MC is an infinite sequence $\rho = s_1 s_2 \ldots$ with $\delta(s_i, \pi(s_i))(s_{i+1}) > 0$. Paths denotes the set of all paths and $\mathbb{P}_s^\pi$ denotes the unique probability measure of $\mathcal{M}^\pi$ over infinite paths starting in the state s.

A *reachability objective* $\mathrm{P}_{\mathsf{opt}}(\mathsf{T})$ with set of target states $\mathsf{T} \subseteq \mathsf{S}$ and $\mathsf{opt} \in \{\max, \min\}$ induces a random variable $X\colon \mathsf{Paths} \to [0,1]$ over paths by assigning 1 to all paths that eventually reach the target and 0 to all others. $\mathrm{E}_{\mathsf{opt}}(\mathsf{rew})$ denotes an *expected reward objective*, where $\mathsf{rew}\colon \mathsf{S} \to \mathbb{Q}_{\geq 0}$ assigns a reward to each state. $\mathsf{rew}(\rho) := \sum_{i=1}^{\infty} \mathsf{rew}(s_i)$ is the accumulated reward of a path $\rho = s_1 s_2 \ldots$. This yields a random variable $X\colon \mathsf{Paths} \to \mathbb{Q} \cup \{\infty\}$ that maps paths to their reward. For a given objective and its random variable X, the *value of a state* $s \in \mathsf{S}$ is the expectation of X under the probability measure $\mathbb{P}_s^\pi$ of the the MC induced by an optimal policy π from the set of all policies Π, formally $\mathsf{V}(s) := \mathsf{opt}_{\pi \in \Pi} \mathbb{E}_s^\pi[X]$.

2.2 Solution Algorithms

Value iteration (VI), e.g. [15], computes a sequence of value vectors converging to the optimum in the limit. In all variants of the algorithm, we start with a function $x\colon \mathsf{S} \to \mathbb{Q}$ that assigns to every state an estimate of the value. The algorithm repeatedly performs an update operation to improve the estimates. After some preprocessing, this operation has a unique fixpoint when $x = \mathsf{V}$. Thus, value iteration converges to the value in the limit. Variants of VI include interval iteration [34], sound VI [56] and optimistic VI [40]. We do not discuss these in detail, but instead refer to the respective papers.

Linear programming (LP), e.g. [6, Chapter 10], encodes the transition structure of the MDP and the objective as a linear optimization problem. For every state, the LP has a variable representing an estimate of its value. Every state-action pair is encoded as a constraint on these variables, as are the target set or rewards. The unique optimum of the LP is attained if and only if for every state its corresponding variable is set to the value of the state. We provide an in-depth discussion of theoretical and practical aspects of LP in Section 3.

Policy iteration (PI), e.g. [11, Section 4], computes a sequence of policies. Starting with an initial policy, we evaluate its induced MC, improve the policy by switching suboptimal choices and repeat the process on the new policy. As every policy improves the previous one and there are only finitely many memoryless deterministic policies (a number exponential in the number of states), eventually we obtain an optimal policy. We further discuss PI in Section 4.

2.3 Guarantees

Given the stakes in many application domains, we require guarantees about the relation between an algorithm's result $\bar{v}$ and the true value v. First, implementations are subject to floating-point errors and imprecision [59] unless they use exact (rational) arithmetic or safe rounding [36]. This can result in arbitrary

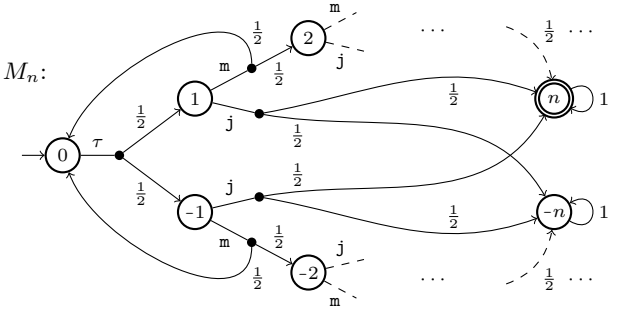


Fig. 1: A hard MDP for all algorithms

Table 1: Correct results

alg.	solver	$n \leq$
PI	–	20
LP	COPT	18
	CPLEX	18
	Glop	25
	GLPK	24
	Gurobi	18
	HiGHS	22
	lp_solve	28
	Mosek	22
	SoPlex	34

differences between $\bar{v}$ and v. Second are the algorithm's inherent properties: VI is an approximating algorithm that converges to the true value only in the limit. In theory, it is possible to obtain the exact result by rounding after exponentially many iterations [15]; in practice, this results in excessive runtime. Instead, for years, implementations used a naive stopping criterion that could return arbitrarily wrong results [33]. This problem's discovery sparked the development of sound variants of VI [8,34,40,19,54,56], including interval iteration, sound value iteration, and optimistic value iteration. A sound VI algorithm guarantees ε-precise results, i.e. $|v - \bar{v}| \leq \varepsilon$ or $|v - \bar{v}| \leq v \cdot \varepsilon$. For LP and PI, the guarantees have not yet been thoroughly investigated. Theoretically, both are exact, but implementations are often not. We discuss the problems in Sections 3 and 4.

The handcrafted MC of [33, Figure 2] highlights the lack of guarantees of VI: standard implementations return vastly incorrect results. We extended it with action choices to obtain the MDP M_n shown in Fig. 1 for $n \in \mathbb{N}$, $n \geq 2$. It has $2n+1$ states; we compute $\mathrm{P}_{\min}(\{n\})$ and $\mathrm{P}_{\max}(\{n\})$. The policy that chooses action `m` wherever possible induces the MC of [33, Figure 2] with $(\mathrm{P}_{\min}(\{n\}), \mathrm{P}_{\max}(\{n\})) = (\frac{1}{2}, \frac{1}{2})$. In every state s with $0 < s < n$, we added the choice of action `j` that jumps to n and $-n$. With that, the (optimal) values over all policies are $(\frac{1}{3}, \frac{2}{3})$. In VI, starting from value 0 for all states except n, initially taking `j` everywhere looks like the best policy for $\mathrm{P}_{\max}$. As updated values slowly propagate, state-by-state, `m` becomes the optimal choice in all states except $-n+1$. We thus layered a "deceptive" decision problem on top of the slow convergence of the original MC. For $n = 20$, VI with Storm and mcsta deliver the incorrect results $(0.247, 0.500)$. For Storm's PI and various LP solvers, we show in Table 1 the largest n for which they return a ± 0.01-correct result. For larger n, PI and all LP solvers claim $\approx (\frac{1}{2}, \frac{1}{2})$ as the correct solution except for Glop and GLPK which only fail for the maximum at the given n; for the minimum, they return the wrong result at $n \geq 29$ and 52, respectively. Sound VI algorithms and Storm's exact-arithmetic engine produce (ε-)correct results, though the former at excessive runtime for larger n. We used default settings for all tools and solvers.

2.4 Optimizations

VI, LP, and PI can all benefit from the following optimizations:

Graph-theoretic algorithms can be used for qualitative analysis of the MDP, i.e. finding states with value 0 or (only for reachability objectives) 1. These qualitative approaches are typically a lot faster than the numerical computations for quantitative analysis. Thus, we always apply them first and only run the numerical algorithms on the remaining states with non-trivial values.

Topological methods, e.g. [17], do not consider the whole MDP at once. Instead, they first compute a topological ordering of the strongly connected components (SCCs)[5] and then analyze each SCC individually. This can improve the runtime, as we decompose the problem into smaller subproblems. The subproblems can be solved with any of the solution methods. Note that when considering acyclic MDPs, the topological approach does not need to call the solution methods, as the resulting values can immediately be backpropagated.

Collapsing of maximal end components (MECs), e.g., [13,34], transforms the MDP into one with equivalent values but simpler structure. After collapsing MECs, the MDP is contracting, i.e. we almost surely reach a target state or a state with value zero. VI algorithms rely on this property for convergence [34,40,56]. For PI and LP, simplifying the graph structure before applying the solution method can speed up the computation.

Warm starts, e.g. [26,46], may adequately initialize an algorithm, i.e., we may provide it with some prior knowledge so that the computation has a good starting point. We implement warm starts by first running VI for a limited number of iterations and using the resulting estimate to guess bounds on the variables in an LP or a good initial policy for PI. See Sections 3 and 4 for more details.

3 Practically solving MDPs using Linear Programs

This section considers the LP-based approach to solving the optimal policy problem in MDPs. To the best of our knowledge, this is the only polynomial-time approach. We discuss various configurations. These configuration are a combination of the LP formulation, the choice of software, and their parameterization.

3.1 How to encode MDPs as LPs?

For objective $\mathrm{P}_{\max}(\mathsf{T})$ we formulate the following LP over variables x_s, $s \in \mathsf{S} \setminus \mathsf{T}$:

$$\text{minimize} \quad \sum_{s \in \mathsf{S}} x_s \quad \text{s.t. } lb(s) \leq x_s \leq ub(s) \quad \text{and}$$
$$x_s \geq \sum_{s' \in \mathsf{S} \setminus \mathsf{T}} \delta(s,a)(s') \cdot x_{s'} + \sum_{t \in \mathsf{T}} \delta(s,a)(t) \quad \text{for all } s \in \mathsf{S} \setminus \mathsf{T}, a \in \mathsf{A}$$

[5] A set $\mathsf{S}' \subseteq \mathsf{S}$ is a connected component if for all $s, s' \in \mathsf{S}'$, s can be reached from s'. We call S' strongly connected component if it is inclusion maximal.

We assume bounds $lb(s) = 0$ and $ub(s) = 1$ for $s \in \mathsf{S} \setminus \mathsf{T}$. The unique solution $\eta\colon \{ x_s \mid s \in \mathsf{S} \setminus \mathsf{T} \} \to [0,1]$ to this LP coincides with the desired objective values $\eta(x_s) = V(s)$. Objectives $\mathrm{P}_{\min}(\mathsf{T})$ and $\mathrm{E}_{\mathsf{opt}}(\mathsf{rew})$ have similar encodings: minimizing policies require maximisation in the LP and flipping the constraint relation. Rewards can be added as an additive factor on the right-hand side. For practical purposes, the LP formulation can be tweaked.

The choice of bounds. Any bounds that respect the unique solution will not change the answer. That is, any lb and ub with $0 \leq lb(s) \leq V(s) \leq ub(s)$ yield a sound encoding. While these additional bounds are superfluous, they may significantly prune the search space. We investigate trivial bounds, e.g., knowing that all probabilities are in $[0,1]$, bounds from a structural analysis as discussed by [8], and bounds induced by a warm start of the solver. For the latter, if we have obtained values $V' \leq V$, e.g., induced by a suboptimal policy, then $V'(s)$ is a lower bound on the value x_s, which is particularly relevant as the LP minimizes.

Equality for unique actions. Markov chains, i.e., MDPs where $|\mathsf{A}| = 1$, can be solved using linear equation systems. The LP encoding uses one-sided inequalities and the objective function to incorporate nondeterministic choices. We investigate adding constraints for all states with a unique action.

$$x_s \leq \sum_{s' \in S \setminus T} \delta(s,a)(s') \cdot x_{s'} + \sum_{t \in T} \delta(s,a)(t) \quad \text{for all } s \in S \setminus T \text{ with } \mathsf{A}(s) = \{a\}$$

These additional constraints may trigger different optimizations in a solver, e.g., some solvers use Gaussian elimination for variable elimination.

A simpler objective. The standard objective assures the solution η is optimal for *every* state, whereas most invocations require only optimality in some specific states – typically the initial state s_0 or the entry states of a strongly connected component. In that case, the objective may be simplified to optimize only the value for those states. This potentially allows for multiple optimal solutions: in terms of the MDP, it is no longer necessary to optimize the value for states that are not reached under the optimal policy.

Encoding the dual formulation. Encoding a dual formulation to the LP is interesting for mixed-integer extensions to the LP, relevant for computing, e.g., policies in POMDPs [47], or when computing minimal counterexamples [58]. For LPs, due to the strong duality, the internal representation in the solvers we investigated is (almost) equivalent and all solvers support both solving the primal and the dual representation. We therefore do not further consider constructing them.

3.2 How to solve LPs with existing solvers?

We rely on the performance of state-of-the-art LP solvers. Many solvers have been developed and are still actively advanced, see [2] for a recent comparison on general benchmarks. We list the LP solvers that we consider for this work in Table 2. The columns summarize for each solver the type of license, whether it uses exact or floating-point arithmetic, whether it supports multithreading,

Table 2: Available LP solvers ("intr = interior point)

solver	version	license	exact/fp	parallel	algorithms	mcsta	Storm
COPT [24]	5.0.5	academic	fp	yes	intr + simplex	yes	no
CPLEX [44]	22.10	academic	fp	yes	intr + simplex	yes	no
Gurobi [32]	9.5	academic	fp	yes	intr + simplex	yes	yes
GLPK [29]	4.65	GPL	fp	no	intr + simplex	no	yes
Glop [30]	9.4.1874	Apache	fp	no	simplex only	yes	no
HiGHS [35,43]	1.2.2	MIT	fp	yes	intr + simplex	yes	no
lp_solve [10]	5.5.2.11	LGPL	fp	no	simplex only	yes	no
Mosek [52]	10.0	academic	fp	yes	intr + simplex	yes	no
SoPlex [28]	6.0.1	academic	both	no	simplex only	no	yes
Z3 [53]	4.8.13	MIT	exact	no	simplex only	no	yes

and what type of algorithms it implements. We also list whether the solver is available from the two model checkers used in this study[6].

Methods. We briefly explain the available methods and refer to [12] for a thorough treatment. Broadly speaking, the LP solvers use one out of two families of methods. *Simplex*-based methods rely on highly efficient pivot operations to consider vertices of the simplex of feasible solutions. Simplex can be executed either in the *primal* or *dual* fashion, which changes the direction of progress made by the algorithm. Our LP formulation has more constraints than variables, which generally means that the dual version is preferable. *Interior methods*, often the subclass of *barrier methods*, do not need to follow the set of vertices. These methods may achieve polynomial time worst-case behaviour. It is generally claimed that simplex has superior average-case performance but is highly sensitive to perturbations, while interior-point methods have a more robust performance.

Warm starts. LP-based model checking can be done using two types of warm starts. Either by providing a (feasible) basis point as done in [26] or by presenting bounds. The former, however, comes with various remarks and limitations, such as the requirement to disable preprocessing. We therefore used warm starts only by using bounds as discussed above.

Multithreading. We generally see two types of parallelisation in LP solvers. Some solvers support a *portfolio* approach that runs different approaches and finishes with the first one that yields a result. Other solvers parallelize the interior-point and/or simplex methods themselves.

Guarantees for numerical LP solvers. All LP solvers allow tweaking of various parameters, including *tolerances* to manage whether a point is considered feasible or optimal, respectively. The experiments in Table 1 already indicate that these guarantees are *not* absolute. A limited experiment indicated that reducing these tolerances towards zero did remove some incorrect results, but not all.

[6] Support for Gurobi, GLPK, and Z3 was already available in Storm. Support for Glop was already available in mcsta. All other solver interfaces have been added.

Exact solving. SoPlex supports exact computations, with a Boost library wrapping GMP rationals [22], after a floating-point arithmetic-based startup phase [27]. While this combination is beneficial for performance in most settings, it leads to crashes for the numerically challenging models. Z3 supports only exact arithmetic (also wrapping GMP numbers with their own interface). We observe that the price of converting large rational numbers may be substantial. SMT solvers like Z3 use a simplex variation [18] tailored towards finding feasible points and in an incremental fashion, optimized for problems with a nontrivial Boolean structure. In contrast, our LP formulation is easily feasible and is a pure conjunction.

4 Sound Policy Iteration

Starting with an initial policy, PI-based algorithms iteratively improve the policy based on the values obtained for the induced MC. The algorithm for solving the induced MC crucially affects the performance and accuracy of the overall approach. This section addresses the solvers available in Storm, possible precision issues, and how to utilize a warm start, while Section 5 discusses PI performance[7].

Markov chain solvers. To solve the induced MC, Storm can employ all linear equation solvers listed in [42] and all implemented variants of VI. In our experiments, we consider (i) the generalized minimal residual method (GMRES) [57] implemented in GMM++ [25], (ii) VI [15] with a standard (relative) termination criterion, (iii) optimistic VI (OVI) [40], and (iv) the sparse LU decomposition implemented in Eigen [31] using either floating-point or exact arithmetic (LU^{X}). LU and LU^{X} provide exact results (modulo floating-point errors in LU) while OVI yields ε-precise results. VI and GMRES do not provide any guarantees.

Correctness of PI. The accuracy of PI is affected by the MC solver. Firstly, PI cannot be more precise than its underlying solver: the result of PI has the same precision as the result obtained for the final MC. Secondly, inaccuracies by the solver can hide policy improvements; this may lead to premature convergence with a sub-optimal policy. We show that PI can return arbitrarily wrong results—*even if the intermediate results are ε-precise*:

Consider the MDP in Fig. 2 with objective $\text{P}_{\max}(\{G\})$. There is only one nondeterministic choice, namely in state s_0. The optimal policy is to pick b, obtaining a value of 0.5. Picking a only yields 0.1. However, when starting from the initial policy $\pi(s_0) = \text{a}$, an ε-precise MC solver may return $0.1 + \varepsilon$ for both s_0 and s_1 and $\delta/2 + (1-\delta) \cdot 0.1$ for s_2. This solution is indeed ε-precise. However, when evaluating which action to pick in s_0, we can choose δ such that a seems to obtain a higher value. Concretely, we require $\delta/2 + (1-\delta) \cdot 0.1 < 0.1 + \varepsilon$. For every $\varepsilon > 0$, this can be achieved by setting $\delta < 2.5 \cdot \varepsilon$. In this case, PI would terminate with the final policy inducing a severely suboptimal value.

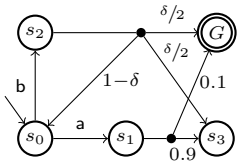


Fig. 2: Example MDP

[7] [46] addresses performance in the context of PI for stochastic games.

If every Markov chain is solved precisely, PI is correct. Indeed, it suffices to be certain that one action is better than all others. This is the essence of modified policy iteration as described in [55, Chapters 6.5 and 7.2.6]. Similarly, [46, Section 4.2] suggests to use interval iteration when solving the system induced by the current policy and stopping when the under-approximation of one action is higher than the over-approximation of all other actions.

Warm starts. PI profits from being provided a *good* initial policy. If the initial policy is already optimal, PI terminates after a single iteration. We can inform our choice of the initial policy by providing estimates for all states as computed by VI. For every state, we choose the action that is optimal according to the estimate. This is a good way to leverage VI's ability to quickly deliver good estimates [40], while at the same time providing the exactness guarantees of PI.

5 Experimental Evaluation

To understand the practical performance of the different algorithms, we performed an extensive experimental evaluation. We used three sets of benchmarks: all applicable benchmark instances[8] from the Quantitative Verification Benchmark Set (QVBS) [41] (the *qvbs* set), a subset of hard QVBS instances (the *hard* set), and numerically challenging models from a runtime monitoring application [45] (the *premise* set, named for the corresponding prototype). We consider two probabilistic model checkers, Storm [42] and the Modest Toolset's [37] mcsta. We used Intel Xeon Platinum 8160 systems running 64-bit CentOS Linux 7.9, allocating 4 CPU cores and 32 GB RAM to each experiment unless noted otherwise.

We plot algorithm runtimes in seconds in *quantile plots* as on the left and *scatter plots* as on the right of Fig. 3. The former compare multiple tools or configurations; for each, we sort the instances by runtime and plot the corresponding monotonically increasing line. Here, a point (x, y) on the a-line means that the x-th fastest instance solved by a took y seconds. The latter compare two tools or configurations. Each point (x, y) is for one benchmark instance: the x-axis tool took x while the y-axis tool took y seconds to solve it. The shape of points indicates the model type; the mapping from shapes to types is the same for all scatter plots and is only given explicitly in the first one in Fig. 3. Additional plots to support the claims in this section are provided in the appendix of the full version [39] of this paper.

The depicted runtimes are for the respective algorithm and all necessary and/or stated preprocessing, but do not include the time for constructing the MDP state spaces (which is independent of the algorithms). mcsta reports all time measurements rounded to multiples of 0.1 s. We summarize timeouts, out-of-memory, errors, and incorrect results as "n/a". Our timeout is 30 minutes for the algorithm and 45 minutes for total runtime including MDP construction. We consider a result $\bar{v}$ incorrect if $|v - \bar{v}| > v \cdot 10^{-3}$ (i.e. relative error 10^{-3}) whenever a reference result v is available. We however do not flag a result as incorrect if

[8] A *benchmark instance* is a combination of model, parameter valuation, and objective.

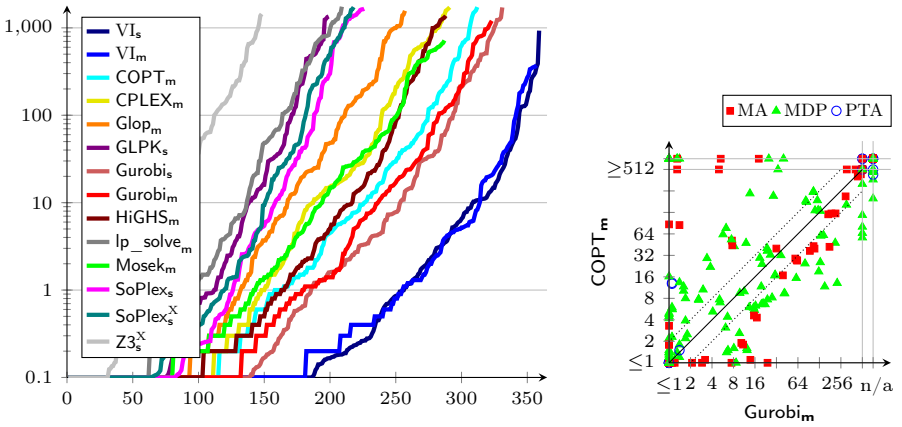


Fig. 3: Comparison of LP solver runtime on the *qvbs* set

v and $\bar{v}$ are both below 10^{-8} (relevant for the *premise* set). Nevertheless, we configure the (unsound) convergence threshold for VI as 10^{-6} relative; among the sound VI algorithms, we include OVI, with a (sound) stopping criterion of relative 10^{-6} error. To only achieve the 10^{-3} precision we actually test, OVI could thus be even faster than it appears in our plots. We make this difference to account for the fact that many algorithms, including the LP solvers, do not have a sound error criterion. We mark exact algorithms/solvers that use rational arithmetic with a superscript $^{\mathrm{X}}$. The other configurations use floating-point arithmetic (fp).

5.1 The QVBS Benchmarks

The *qvbs* set comprises all QVBS benchmark instances with an MDP, Markov automaton (MA), or probabilistic timed automaton (PTA) model[9] and a reachability or expected reward/time objective that is quantitative, i.e. not a query that yields a zero or one probability. We only consider instances where both Storm and mcsta can build the explicit representation of the MDP within 15 minutes. This yields 367 instances. We obtain reference results for 344 of them from either the QVBS database or by using one of Storm's exact methods. We found all reference results obtained via different methods to be consistent.

For LP, we have various solvers with various parameters each, cf. Section 3. For conciseness, we first compare all available LP solvers on the *qvbs* set. For the best-performing solver, we then evaluate the benefit of different solver configurations. We do the same for the choice of Markov chain solution method in PI. We then focus on these single, reasonable, setups for LP and PI each in more detail.

LP solver comparison. The left-hand plot of Fig. 3 summarizes the results of our comparison of the different LP solvers. Subscripts $_s$ and $_m$ indicate whether the solver is embedded in either Storm or mcsta. We apply no optimizations or

[9] MA and PTA are converted to MDP via embedding and digital clocks [48].

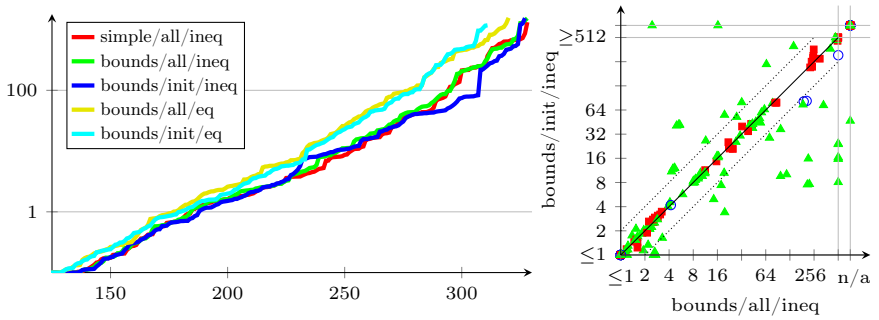


Fig. 4: Performance impact of LP problem formulation variants (using Gurobi$_s$)

reductions to the MDPs except for the precomputation of probability-0 states (and in Storm also of probability-1 states), and use the default settings for all solvers, with the trivial variable bounds $[0, 1]$ and $[0, \infty)$ for probabilities and expected rewards, respectively. We include VI as baseline. In Table 3, we summarize the results.

Table 3: LP summary

solver	correct	incorr.	no result
VI$_s$	359	8	0
VI$_m$	357	8	2
COPT$_m$	312	12	43
CPLEX$_m$	291	10	66
G op$_m$	257	4	106
GLPK$_s$	199	5	163
Gurobi$_s$	331	4	32
Gurobi$_m$	323	4	40
HiGHS$_m$	288	10	69
p_so ve$_m$	209	0	158
Mosek$_m$	287	15	65
SoP ex$_s$	226	9	132
SoP ex$_s^X$	218	0	149
Z3$_s^X$	148	0	219

In terms of **performance** and scalability, Gurobi solves the highest number of benchmarks in any given time budget, closely followed by COPT. CPLEX, HiGHS, and Mosek make up a middle-class group. While the exact solver Z3 is very slow, SoPlex's exact mode actually competes with some fp solvers. However, the quantile plots do not tell the whole story. On the right of Fig. 3, we compare COPT and Gurobi directly: each has a large number of instances on which it is (much) better.

In terms of **reliability** of results, the exact solvers as expected produce no incorrect results; so does the slowest fp solver, lp_solve. COPT, CPLEX, HiGHS, Mosek, and fp-SoPlex perform badly in this metric, producing more errors than VI. Interestingly, these are mostly the faster solvers, the exception being Gurobi.

Overall, Gurobi achieves highest performance at decent reliability; in the remainder of this section, we thus use Gurobi$_s$ whenever we apply non-exact LP.

LP solver tweaking. Gurobi can be configured to use an "*auto*" portfolio approach, potentially running multiple algorithms concurrently on multiple threads, a primal or a dual simplex algorithm, or a barrier method algorithm. We compared each option with 4 threads and found no significant performance difference. Similarly, running the *auto* method with 1, 4, and 16 threads (only here, we allocate 16 threads per experiment) also failed to bring out noticeable performance differences. Using more threads results in a few more out-of-memory errors, though. We thus fix Gurobi on *auto* with 4 threads.

Fig. 4 shows the performance impact of supplying Gurobi with more precise bounds on the variables for expected reward objectives using methods from

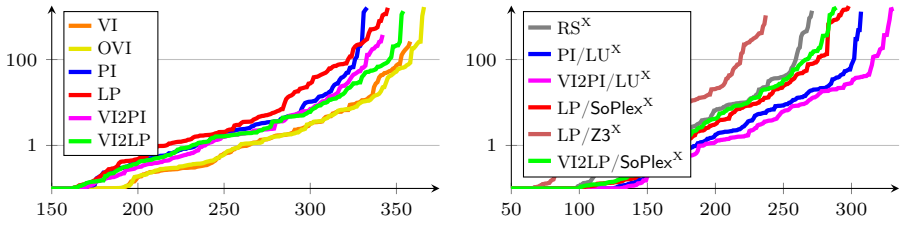
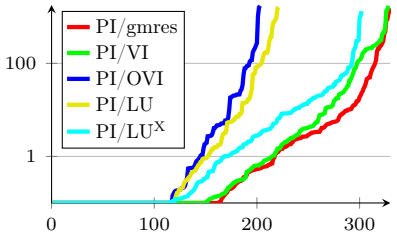


Fig. 5: Comparison of MDP model checking algorithms on the *qvbs* set

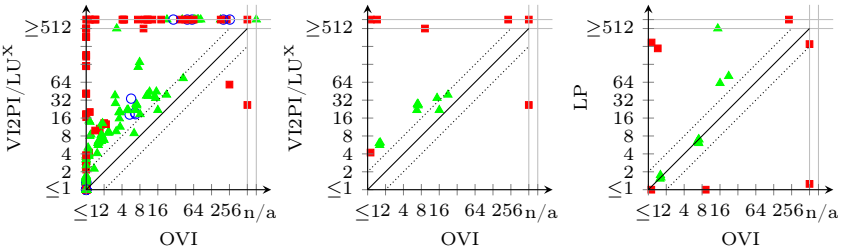
[8,51] ("bounds" instead of "simple"), of optimizing only for initial state ("init") instead of the sum over all states ("all"), and of using equality ("eq") instead of less-/greater-than-or-equal ("ineq") for unique action states. More precise bounds yield a very small improvement at essentially no cost. Optimizing for the initial state only results in a little better overall performance (in the "pocket" in the quantile plot around $x = 315$ that is also clearly visible in the scatter plot). However, it also results in 2 more incorrect results in the *qvbs* set. Using equality for unique actions noticeably decreases performance and increases the incorrect result count by 9 instances. For all experiments that follow, we thus use the more precise bounds, but do not enable the other two optimizations.



PI methods comparison. The main choice in PI is which algorithm to use to solve the induced Markov chains. On the right, we show the performance of the different algorithms available in Storm (cf. Section 4). LU^{X} yields a fully exact PI. This interestingly performs better than the fp version, potentially because fp errors induce spurious policy changes. The same effect likely also hinders the use of OVI, whereas VI leads to good performance. Nevertheless, gmres is best overall, and thus our choice for all following experiments with non-exact PI. VI and gmres yield 6 and 4 incorrect results, respectively. OVI and the exact methods are always correct on this benchmark set.

Best MDP algorithms for QVBS. We now compare all MDP model checking algorithms on the *qvbs* set: with floating-point numbers, LP and PI configured as described above, plus unsound VI, sound OVI, and the warm-start variants of PI and LP denoted "VI2PI" and "VI2LP", respectively. Exact results are provided by rational search (RS, essentially an exact version of VI) [50], PI with exact LU, and LP with exact solvers (SoPlex and Z3). All are implemented in Storm.

In a first experiment, we evaluated the impact of using the topological approach and of collapsing MECs (cf. Section 2.4). The results, for which we omit plots, are that the topological approach noticeably improves performance and scalability for *all* algorithms, and we therefore always use it from now on. Collapsing MECs is necessary to guarantee termination of OVI, while for the



(a) *qvbs* OVI vs. VI2PI^{X} (b) *hard* OVI vs. VI2PI^{X} (c) *hard* OVI vs. LP

Fig. 6: Additional direct performance comparisons

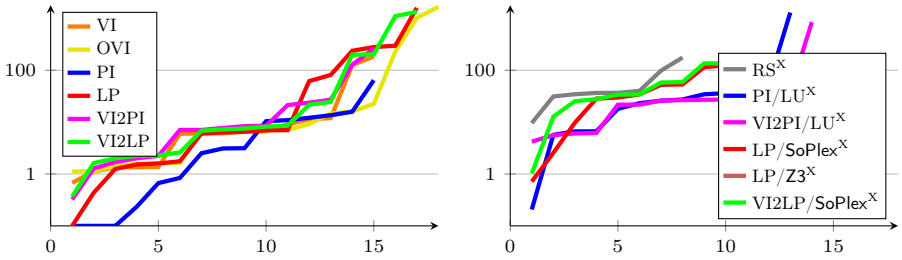


Fig. 7: Comparison of MDP model checking algorithms on the *hard* subset

other algorithms it is a potential optimization; however we found it to overall have a minimal positive performance impact only. Since it is required by OVI and does not reduce performance, we also always use it from now on.

Fig. 5 shows the complete comparison of all the methods on the *qvbs* set, for fp algorithms on the left and exact solutions on the right. Among the fp algorithms, OVI is clearly the fastest and most scalable. VI is somewhat faster but incurs several incorrect results that diminish its appearance in the quantile plot. OVI is additionally special among these algorithms in that it is sound, i.e. provides guaranteed ε-correct results—though up to fp rounding errors, which can be eliminated following the approach of [36]. On the exact side, PI with an inexact-VI warm start works best. The scatter plots in Fig. 6(a) shows the performance impact of computing an exact instead of an approximate solution.

5.2 The Hard QVBS Benchmarks

The QVBS contains many models built for tools that use VI as default algorithm. The other algorithms may actually be important to solve key challenging instances where VI/OVI perform badly. This contribution could be hidden in the sea of instances trivial for VI. We thus zoom in on a selection of QVBS instances that appear "hard" for VI: those where VI takes longer than the prior MDP state

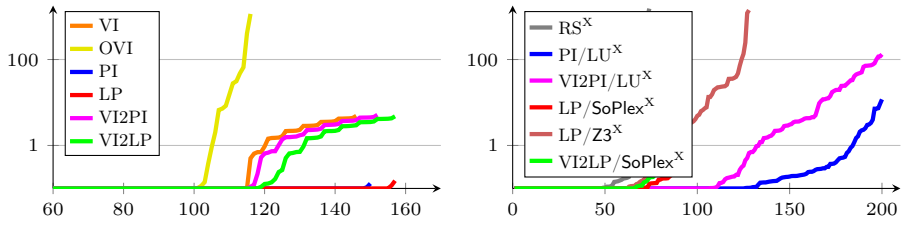


Fig. 8: Comparison of MDP model checking algorithms on the *premise* set

space construction phase in both Storm and mcsta, and additionally both phases together take at least 1 s. These are 18 of the previously considered 367 instances.

In Fig. 7, we show the behaviour of all the algorithms on this *hard* subset. OVI again works better than VI due to the incorrect results that VI returns. We see that the performance and scalability gap between the algorithms has narrowed; although OVI still "wins", LP in particular is much closer than on the full *qvbs* set. We also investigated the LP outcomes with solvers other than Gurobi: even on this set, Gurobi and COPT remain the fastest and most scalable solvers. With mcsta, in the basic configuration, they solve 16 and 17 instances, the slowest taking 835 s and 1334 s, respectively; with the topological optimization, the numbers become 17 and 15 instances with the slowest at 1373 s and 1590 s seconds. We show the detailed comparison of OVI and LP in Fig. 6(c), noting that there are a few instances where LP is much faster, and repeat the comparison between the best fp and exact algorithms (Fig. 6(b)).

5.3 The Runtime Monitoring Benchmarks

While the QVBS is intentionally diverse, our third set of benchmarks is intentionally focused: We study 200 MDPs from a runtime monitoring study [45]. The original problem is to compute the normalized risk of continuing to operate the system being monitored subject to stochastic noise, unobservable and uncontrollable nondeterminism, and partial state observations. This is a query for a conditional probability. It is answered via probabilistic model checking by unrolling an MDP model along an observed history trace of length $n \in \{50, \ldots, 1000\}$ following the approach of Baier et al. [7]. The MDPs contain many transitions back to the initial state, ultimately resulting in numerically challenging instances (containing structures similar to the one of M_n in Section 2.3). We were able to compute a reference result for all instances.

Fig. 8 compares the different MDP model checking algorithms on this set. In line with the observations in [45], we see very different behaviour compared to the QVBS. Among the fp solutions on the left, LP with Gurobi terminates very quickly (under 1 s), and either produces a correct (155 instances) or a completely incorrect result (mostly 0, on 45 instances). VI behaves similarly, but is slower. OVI, in contrast, delivers no incorrect result, but instead fails to terminate on all but 116 instances. In the exact setting, warm starts using VI inherit its relative

slowness and consequently do not pay off. Exact PI outperforms both exact LP solvers. In the case of exact SoPlex, out of the 112 instances it does not manage to solve, 98 are crashes likely related to a confirmed bug in its current version.

The *premise* set highlights that the best MDP model checking algorithm depends on the application. Here, in the fp case, LP appears best but produces unreliable (incorrect) results; the seemingly much worse OVI at least does not do so. Given the numeric challenge, an exact method should be chosen, and we show that these actually perform well here.

6 Conclusion

We thoroughly investigated the state of the art in MDP model checking, showing that there is no single best algorithm for this task. For benchmarks which are not numerically challenging, OVI is a sensible default, closely followed by PI and LP with a warm start—although using the latter two means losing soundness as confirmed by a number of incorrect results in our experiments. For numerically hard benchmarks, PI and LP as well as computing exact solutions are more attractive, and clearly preferable in combination. Overall, although LP has the superior (polynomial) theoretical complexity, in our practical evaluation, it almost always performs worse than the other (exponential) approaches. This is even though we use modern commercial solvers and tune both the LP encoding of the problem as well as the solvers' parameters. While we *observed* the behaviour of the different algorithms and have some intuition into what makes the *premise* set hard, an entire research question of its own is to identify and quantify the structural properties that make a model hard.

Our evaluation also raises the question of how prevalent MDPs that challenge VI are in practice. Aside from the *premise* benchmarks, we were unable to find further sets of MDPs that are hard for VI. Notably, several stochastic games (SGs) difficult for VI were found in [46]; the authors noted that using PI for the SGs was better than applying VI to the SGs. However, when we extracted the induced MDPs, we found them all easy for VI. Similarly, [3] used a random generation of SGs of at most 10,000 states, many of which were challenging for the SG algorithms. Yet the same random generation modified to produce MDPs delivered only MDPs easily solved in seconds, even with drastically increased numbers of states. In contrast, Alagöz et al. [1] report that their random generation returned models where LP beat PI. However, their setting is discounted, and their description of the random generation was too superficial for us to be able to replicate it. We note that, in several of our scatter plots, the MA instances from the QVBS (where we check the embedded MDP) appeared more challenging overall than the MDPs. We thus conclude this paper with a call for challenging MDP benchmarks—as separate benchmark sets of unique characteristics like *premise*, or for inclusion in the QVBS.

Data availability statement. The datasets generated and analysed in this study and code to regenerate them are available in the accompanying artifact [38]. For Storm, our code builds on version 1.7.0. We used mcsta version 3.1.213.

References

1. Alagöz, O., Ayvaci, M.U.S., Linderoth, J.T.: Optimally solving Markov decision processes with total expected discounted reward function: Linear programming revisited. Comput. Ind. Eng. **87**, 311–316 (2015). https://doi.org/10.1016/j.cie.2015.05.031
2. Anand, R., Aggarwal, D., Kumar, V.: A comparative analysis of optimization solvers. Journal of Statistics and Management Systems **20**(4), 623–635 (2017). https://doi.org/10.1080/09720510.2017.1395182
3. Azeem, M., Evangelidis, A., Kretínský, J., Slivinskiy, A., Weininger, M.: Optimistic and topological value iteration for simple stochastic games. CoRR **abs/2207.14417** (2022). https://doi.org/10.48550/arXiv.2207.14417
4. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Handbook of Model Checking, pp. 963–999. Springer (2018)
5. Baier, C., Hermanns, H., Katoen, J.P.: The 10,000 facets of MDP model checking. In: Computing and Software Science, LNCS, vol. 10000, pp. 420–451. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_21
6. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008), https://mitpress.mit.edu/books/principles-model-checking
7. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: TACAS. LNCS, vol. 8413, pp. 515–530. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_43
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: CAV (1). LNCS, vol. 10426, pp. 160–180. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_8
9. Balaji, N., Kiefer, S., Novotný, P., Pérez, G.A., Shirmohammadi, M.: On the complexity of value iteration. In: 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
10. Berkelaar, M., Eikland, K., Notebaert, P.: Introduction to lp_solve 5.5.2.11, https://lpsolve.sourceforge.net/5.5/, accessed 2023-01-25.
11. Bertsekas, D.P., Tsitsiklis, J.N.: An analysis of stochastic shortest path problems. Math. Oper. Res. **16**(3), 580–595 (1991). https://doi.org/10.1287/moor.16.3.580
12. Boyd, S.P., Vandenberghe, L.: Convex Optimization. Cambridge University Press (2014)
13. Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Kretínský, J., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: ATVA. LNCS, vol. 8837, pp. 98–114. Springer (2014). https://doi.org/10.1007/978-3-319-11936-6_8
14. Budde, C.E., Hartmanns, A., Klauck, M., Kretínský, J., Parker, D., Quatmann, T., Turrini, A., Zhang, Z.: On correctness, precision, and performance in quantitative verification – QComp 2020 competition report. In: ISoLA (4). LNCS, vol. 12479, pp. 216–241. Springer (2020). https://doi.org/10.1007/978-3-030-83723-5_15
15. Chatterjee, K., Henzinger, T.A.: Value iteration. In: 25 Years of Model Checking. LNCS, vol. 5000, pp. 107–138. Springer (2008). https://doi.org/10.1007/978-3-540-69850-0_7
16. Cubuktepe, M., Jansen, N., Junges, S., Katoen, J.P., Topcu, U.: Convex optimization for parameter synthesis in MDPs. IEEE Trans. Autom. Control. (2022). https://doi.org/10.1109/TAC.2021.3133265

17. Dai, P., Mausam, Weld, D.S., Goldsmith, J.: Topological value iteration algorithms. J. Artif. Intell. Res. **42**, 181–209 (2011), https://www.jair.org/index.php/jair/article/view/10725
18. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: CAV. LNCS, vol. 4144, pp. 81–94. Springer (2006)
19. Eisentraut, J., Kelmendi, E., Kretínský, J., Weininger, M.: Value iteration for simple stochastic games: Stopping criterion and learning algorithm. Inf. Comput. **285**(Part), 104886 (2022). https://doi.org/10.1016/j.ic.2022.104886
20. Fearnley, J.: Exponential lower bounds for policy iteration. In: ICALP (2). LNCS, vol. 6199, pp. 551–562. Springer (2010). https://doi.org/10.1007/978-3-642-14162-1_46
21. Forejt, V., Kwiatkowska, M.Z., Parker, D.: Pareto curves for probabilistic model checking. In: ATVA. LNCS, vol. 7561, pp. 317–332. Springer (2012). https://doi.org/10.1007/978-3-642-33386-6_25
22. Free Software Foundation: The GNU Multiple Precision Arithmetic Library, https://gmplib.org/, accessed 2023-01-25.
23. Funke, F., Jantsch, S., Baier, C.: Farkas certificates and minimal witnesses for probabilistic reachability constraints. In: TACAS (1). LNCS, vol. 12078, pp. 324–345. Springer (2020)
24. Ge, D., Huangfu, Q., Wang, Z., Wu, J., Ye, Y.: Cardinal Optimizer (COPT) user guide (2022), https://guide.coap.online/copt/en-doc
25. GetFEM project: Gmm++ Library, https://getfem.org/gmm/, accessed 2023-01-25.
26. Giro, S.: Optimal schedulers vs optimal bases: An approach for efficient exact solving of Markov decision processes. Theor. Comput. Sci. **538**, 70–83 (2014). https://doi.org/10.1016/j.tcs.2013.08.020
27. Gleixner, A.M., Steffy, D.E., Wolter, K.: Improving the accuracy of linear programming solvers with iterative refinement. In: ISSAC. pp. 187–194. ACM (2012)
28. Gleixner, A.M., Steffy, D.E., Wolter, K.: Iterative refinement for linear programming. Tech. Rep. 3, ZIB, Takustr. 7, 14195 Berlin (2016). https://doi.org/10.1287/ijoc.2016.0692
29. GNU Project: GLPK (GNU Linear Programming Kit), http://www.gnu.org/software/glpk/glpk.html
30. Google: Glop – linear optimization, https://developers.google.com/optimization/lp, accessed 2023-01-25.
31. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010), http://eigen.tuxfamily.org
32. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2022), https://www.gurobi.com
33. Haddad, S., Monmege, B.: Reachability in MDPs: Refining convergence of value iteration. In: RP. LNCS, vol. 8762, pp. 125–137. Springer (2014)
34. 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
35. Hall, J., Galabova, I., Gottwald, L., Feldmeier, M.: HiGHS – high performance software for linear optimization, https://www.maths.ed.ac.uk/hall/HiGHS/, accessed 2023-01-25.
36. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic. In: TACAS (2). LNCS, vol. 13244, pp. 41–59. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_3
37. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: TACAS. LNCS, vol. 8413, pp. 593–598. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_51

38. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A Practitioner's Guide to MDP Model Checking Algorithms (Artefact) (2023). https://doi.org/10.5281/zenodo.7509474
39. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms (2023). https://doi.org/10.48550/ARXIV.2301.10197
40. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: CAV (2). LNCS, vol. 12225, pp. 488–511. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_26
41. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS. LNCS, vol. 11427, pp. 344–350. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_20
42. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022). https://doi.org/10.1007/s10009-021-00633-z
43. Huangfu, Q., Hall, J.A.J.: Parallelizing the dual revised simplex method. Math. Program. Comput. **10**(1), 119–142 (2018). https://doi.org/10.1007/s12532-017-0130-5
44. IBM: IBM ILOG CPLEX Optimizer, https://www.ibm.com/analytics/cplex-optimizer, accessed 2023-01-25.
45. Junges, S., Torfah, H., Seshia, S.A.: Runtime monitors for Markov decision processes. In: CAV (2). LNCS, vol. 12760, pp. 553–576. Springer (2021)
46. Kretinsky, J., Ramneantu, E., Slivinskiy, A., Weininger, M.: Comparison of algorithms for simple stochastic games. Inf. Comput. (2022). https://doi.org/10.1016/j.ic.2022.104885
47. Kumar, A., Zilberstein, S.: History-based controller design and optimization for partially observable MDPs. In: ICAPS. vol. 25, pp. 156–164 (2015)
48. 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
49. Littman, M.L., Dean, T.L., Kaelbling, L.P.: On the complexity of solving Markov decision problems. In: UAI. pp. 394–402. Morgan Kaufmann (1995)
50. Mathur, U., Bauer, M.S., Chadha, R., Sistla, A.P., Viswanathan, M.: Exact quantitative probabilistic model checking through rational search. Formal Methods Syst. Des. **56**(1), 90–126 (2020)
51. 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
52. MOSEK ApS: The MOSEK Optimization Suite 10.0.34, https://docs.mosek.com/latest/intro/index.html, accessed 2023-01-25.
53. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
54. Phalakarn, K., Takisaka, T., Haas, T., Hasuo, I.: Widest paths and global propagation in bounded value iteration for stochastic games. In: CAV (2). LNCS, vol. 12225, pp. 349–371. Springer (2020), https://doi.org/10.1007/978-3-030-53291-8_19
55. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887
56. Quatmann, T., Katoen, J.P.: Sound value iteration. In: CAV (1). LNCS, vol. 10981, pp. 643–661. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_37

57. Saad, Y., Schultz, M.H.: Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems. Siam Journal on Scientific and Statistical Computing **7**, 856–869 (1986), https://epubs.siam.org/doi/10.1137/0907058
58. Wimmer, R., Jansen, N., Vorpahl, A., Ábrahám, E., Katoen, J.P., Becker, B.: High-level counterexamples for probabilistic automata. Log. Methods Comput. Sci. **11**(1) (2015)
59. Wimmer, R., Kortus, A., Herbstritt, M., Becker, B.: Probabilistic model checking and reliability of results. In: DDECS. pp. 207–212. IEEE Computer Society (2008). https://doi.org/10.1109/DDECS.2008.4538787
60. Ye, Y.: The simplex and policy-iteration methods are strongly polynomial for the Markov decision problem with a fixed discount rate. Mathematics of Operations Research **36**(4), 593–603 (2011)

Correct Approximation of Stationary Distributions

Tobias Meggendorfer(✉)

Institute of Science and Technology Austria, 3400 Klosterneuburg, Austria
tobias.meggendorfer@ista.ac.at

Abstract. A classical problem for Markov chains is determining their stationary (or steady-state) distribution. This problem has an equally classical solution based on eigenvectors and linear equation systems. However, this approach does not scale to large instances, and iterative solutions are desirable. It turns out that a naive approach, as used by current model checkers, may yield completely wrong results. We present a new approach, which utilizes recent advances in partial exploration and mean payoff computation to obtain a correct, converging approximation.

1 Introduction

Discrete-time Markov chains (MCs) are an elegant and standard framework to describe stochastic processes, with a vast area of applications such as computer science [4], biology [28], epidemiology [13], and chemistry [12], to name a few. In a nutshell, MC comprise a set of states and a transition function, assigning to each state a distribution over successors. The system evolves by repeatedly drawing a successor state from the transition distribution of the current state. This can, for example, model communication over a lossy channel, a queuing network, or populations of predator and prey which grow and interact randomly. For many applications, the *stationary distribution* of such a system is of particular interest. Intuitively, this distribution describes in which states the system is in after an "infinite" number of steps. For example, in a chemical reaction network this distribution could describe the equilibrium states of the mixture.

Traditionally, the stationary distribution is obtained by computing the dominant eigenvector for particular matrices and solving a series of linear equation systems. This approach is appealing in theory, since it is polynomial in the size of the considered Markov chain. Moreover, since linear algebra is an intensely studied field, many optimizations for the computations at hand are known.

In practice, these approaches however often turn out to be insufficient. Real-world models may have millions of states, often ruling out exact solution approaches. As such, the attention turns to iterative methods. In particular, the popular model checker PRISM [21] employs the *power method* (or *power iteration*) to approximate the stationary distribution. Similar to many other problems on Markov chains, such iterative methods have an exponential worst-case, however obtain good results quickly on many models. (Models where iterative methods indeed converge slowly are called *stiff*.) However, as we show in this work, the

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 489–507, 2023.
https:// oi.org/10.1007/978-3-031-30823-9_25

"absolute change"-criterion used by PRISM to stop the iteration is incorrect. In particular, the produced results may be arbitrarily wrong already on a model with only four states. In [14,7] the authors discuss a similar issue for the problem of *reachability*, also rooted in an incorrect absolute change stopping criterion, and provide a solution through converging lower and *upper* bounds. In our case, the situations is more complicated. The convergence of the power method is quite difficult to bound: A good (and potentially tight) a-priori bound is given by the ratio of first and second eigenvalues, which however is as hard to determine as solving the problem itself. In the case of MC, only a crude bound on this ratio can be obtained easily, which gives an exponential bound on the number of iterations required to achieve a given precision. More strikingly, in contrast to reachability, there is to our knowledge no general *adaptive* stopping criterion for power iteration, i.e. a way to check whether the current iterates are already close to the correct result. Thus, one would always need to iterate for as many steps as given by the a-priori bound to obtain guarantees on the result. In summary, exact solution approaches do not scale well, and the existing iterative approach may yield wrong results or requires an intractable number of steps.

Another, orthogonal issue of the mentioned approaches is that they construct the *complete* system, i.e. determine the stationary distribution for each state. However, when we figure out that, for example, the stationary distribution has a value of at least 99% for one state, all other states can have at most 1% in total. In case we are satisfied with an *approximate* solution, we could already stop the computation here, without investigating any other state. Inspired by the results of [7,18], we thus also want to find such an approximate solution, capable of identifying the relevant parts of the system and only constructing those.

1.1 Contributions

In this work, we address all the above issues. To this end, we

- provide a characterization of the stationary distribution through mean payoff which allows us to obtain provably correct approximations (Section 3),
- introduce a general framework to approximate the stationary distribution in Markov chains, capable of utilizing partial exploration approaches (Section 4),
- as the main technical contribution, provide very general, precise correctness and termination proofs, requiring only minimal assumptions (Theorem 3),
- instantiate this framework with both the classical solution approach as well as our novel sampling-based interval approximation approach (Section 4.2),
- evaluate the variants of our framework experimentally (Section 5), and
- demonstrate with a minimal example that the standard approach of PRISM may yield arbitrarily wrong results (Fig. 2).

1.2 Related Work

Most related is the work of [30], which also try to identify the most relevant parts of the system, however they employ the special structure given by cellular processes to find these regions and estimate the subsequent approximation

error. Many other works deal with special cases, such as queueing models [1,17], time-reversible chains [8], or positive rows (all states have a transition to one particular state) [9,11,27]. In contrast, our methods aim to deal with general Markov chains. We highlight that for the "positive row" case, [11] also provides converging bounds, however through a different route. Another topic of interest are continuous time Markov chains, where abstraction- and truncation-based algorithms are applicable [20,3] and computation of the stationary distribution can be used for time-bounded reachability [16].

2 Preliminaries

As usual, $\mathbb{N}$ and $\mathbb{R}$ refer to the (positive) natural numbers and real numbers, respectively. For a set S, $\overline{S}$ denotes its complement, while $S^\star$ and S^ω refer to the set of finite and infinite sequences comprising elements of S, respectively. We write $\mathbb{1}_S(s) = 1$ if $s \in S$ and 0 otherwise for the *characteristic function* of S.

We assume familiarity with basic notions of probability theory, e.g., *probability spaces*, *probability measures*, and *measurability*; see e.g. [6] for a general introduction. A *probability distribution* over a countable set X is a mapping $d : X \to [0,1]$, such that $\sum_{x \in X} d(x) = 1$. Its *support* is denoted by $\mathrm{supp}(d) = \{x \in X \mid d(x) > 0\}$. $\mathcal{D}(X)$ denotes the set of all probability distributions on X. Some event happens *almost surely* (a.s.) if it happens with probability 1.

The central object of interest are Markov chains, a classical model for systems with stochastic behaviour: A (discrete-time time-homogeneous) *Markov chain (MC)* is a tuple $\mathsf{M} = (S, \delta)$, where S is a finite set of *states*, and $\delta : S \to \mathcal{D}(S)$ is a *transition function* that for each state s yields a probability distribution over successor states. We deliberately exclude the explicit definition of an initial state. We direct the interested reader to, e.g., [4, Sec. 10.1], [29, App. A], or [19] for further information on Markov chains and related notions.

For ease of notation, we write $\delta(s, s')$ instead of $\delta(s)(s')$, and, given a function $f : S \to \mathbb{R}$ mapping states to real numbers, we write $\delta(s)\langle f \rangle \coloneqq \sum_{s' \in S} \delta(s, s') \cdot f(s')$ to denote the weighted sum of f over the successors of s.

We always assume an arbitrary but fixed numbering of the states and identify a state with its respective number. For example, given a vector $v \in \mathbb{R}^{|S|}$ and a state $s \in S$, we may write $v[s]$ to denote the value associated with s by v. In this way, a function $v : S \to \mathbb{R}$ is equivalent to a vector $v \in \mathbb{R}^{|S|}$.

For a set of states $R \subseteq S$ where no transitions leave R, i.e. $\delta(s, s') = 0$ for all $s \in R$, $s' \in S \setminus R$, we define the *restricted Markov chain* $\mathsf{M}|_R \coloneqq (R, \delta|_R)$ with $\delta|_R : R \to \mathcal{D}(R)$ copying the values of δ, i.e. $\delta|_R(s, s') = \delta(s, s')$ for all $s, s' \in R$.

Paths An *infinite path* ρ in a Markov chain is an infinite sequence $\rho = s_1 s_2 \cdots \in S^\omega$, such that for every $i \in \mathbb{N}$ we have that $\delta(s_i, s_{i+1}) > 0$. We use $\rho(i)$ to refer to the i-th state s_i in a given infinite path. We denote the set of all infinite paths of a Markov chain M by $\mathsf{Paths}_\mathsf{M}$. Observe that in general $\mathsf{Paths}_\mathsf{M}$ is a proper subset of S^ω, as we imposed additional constraints. A Markov chain together with an initial state $\hat{s} \in S$ induces a unique probability measure $\mathsf{Pr}_{\mathsf{M},\hat{s}}$ over infinite paths [4, Sec. 10.1]. Given a measurable random variable $f : \mathsf{Paths}_\mathsf{M} \to \mathbb{R}$, we write $\mathbb{E}_{\mathsf{M},\hat{s}}[f] \coloneqq \int_{\rho \in \mathsf{Paths}} f(\rho)\, d\mathsf{Pr}_{\mathsf{M},\hat{s}}$ to denote its expectation w.r.t. this measure.

Reachability An important tool in the following is the notion of *reachability probability*, i.e. the probability that the system, starting from a state $\hat{s}$, will eventually reach a given set T. Formally, for a Markov chain M and set of states T, we define the set of runs which reach T (i) at step n by $\Diamond^{=n} T := \{\rho \in \mathsf{Paths}_{\mathsf{M}} \mid \rho(n) \in T\}$ and (ii) eventually by $\Diamond T = \bigcup_{i=1}^{\infty} \Diamond^{=i} T$. (For a measurability proof see e.g. [4, Chp. 10].) For a state $\hat{s}$, the probability to reach T is given by $\mathsf{Pr}_{\mathsf{M},\hat{s}}[\Diamond T]$.

Classically, the reachability probability can be determined by solving a linear equation system, as follows. For a fixed target set T, let S_0 be all states that cannot reach T. Note that S_0 can be determined by simple graph analysis. Then, the reachability probability $\mathsf{Pr}_{\mathsf{M},\hat{s}}[\Diamond T]$ is the unique solution of [4, Thm. 10.19]

$$f(s) = 1 \text{ if } s \in T, \quad 0 \text{ if } s \in S_0, \quad \text{and} \quad \delta(s)\langle f \rangle \text{ otherwise.} \tag{1}$$

Value Iteration A classical tool to deal with Markov chains is *value iteration* (VI) [5]. It is a simple yet surprisingly efficient and extendable approach to solve a variety of problems. At its heart, VI relies, as the name suggests, on iteratively applying an operation to a value vector. This operation often is called "Bellman backup" or "Bellman update", usually derived from a fixed-point characterization of the problem at hand. Thus, VI often can be viewed as fixed point iteration. For reachability, inspired by Eq. (1), we start from $v_1[s] = 0$ and iterate

$$v_{k+1}[s] = 1 \text{ if } s \in T, \quad 0 \text{ if } s \in S_0, \quad \text{and} \quad \delta(s)\langle v_k \rangle \text{ otherwise.} \tag{2}$$

This iteration monotonically converges to the true value in the limit from below [4, Thm. 10.15], [29, Thm. 7.2.12]. Convergence up to a given precision may take exponential time [14, Thm. 3], but in practice VI often is much faster than methods based on equation solving. For further details, see [26, App. A.2].

Strongly Connected Components A non-empty set of states $C \subseteq S$ in a Markov chain is *strongly connected* if for every pair $s, s' \in C$ there is a non-empty finite path from s to s'. Such a set C is a *strongly connected component* (SCC) if it is inclusion maximal, i.e. there exists no strongly connected C' with $C \subsetneq C'$. SCCs are disjoint, each state belongs to at most one SCC. An SCC is *bottom* (BSCC) if additionally no path leads out of it, i.e. for all $s \in C, s' \in S \setminus C$ we have $\delta(s, s') = 0$. The set of BSCCs in an MC M is denoted by $\mathrm{BSCC}(\mathsf{M})$ and can be determined in linear time by, e.g., Tarjan's algorithm [32].

The bottom components fully capture the limit behaviour of any Markov chain. Intuitively, the following statement says that (i) with probability one a run of a Markov chain eventually forever remains inside one single BSCC, and (ii) inside a BSCC, all states are visited infinitely often with probability one.

Lemma 1 ([4, Thm. 10.27]). *For any MC* M *and state* s*, we have*

$$\mathsf{Pr}_{\mathsf{M},s}[\{\rho \mid \exists R_i \in \mathrm{BSCC}(\mathsf{M}). \exists n_0 \in \mathbb{N}. \forall n > n_0. \rho(n) \in R_i\}] = 1.$$

For any BSCC $R \in \mathrm{BSCC}(\mathsf{M})$ *and states* $s, s' \in R$*, we have* $\mathsf{Pr}_{\mathsf{M},s}[\Diamond\{s'\}] = 1$*.*

Stationary Distribution Given a state $\hat{s}$, the *stationary distribution* (also known as *steady-state* or *long-run distribution*) of a Markov chain intuitively describes, for each state s, the probability for the system to be at this particular state at an

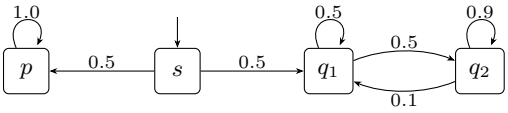


Fig. 1. Example MC to demonstrate the stationary distribution. We have that $\pi^{\infty}_{\mathsf{M},s} = \{p \mapsto \frac{1}{2}, s \mapsto 0, q_1 \mapsto \frac{1}{2} \cdot \frac{1}{6}, q_2 \mapsto \frac{1}{2} \cdot \frac{5}{6}\}$.

arbitrarily chosen step "at infinity". There are several ways to define this notion. In particular, there is a subtle difference between the *limiting* and *stationary distribution*, which however coincide for *aperiodic* MC. For the sake of readability, we omit this distinction and assume w.l.o.g. that all MCs we deal with are aperiodic. See [26, App. A.1] for further discussion. Our definition follows the view of [4, Def. 10.79]; see [29, Sec. A.4] for a different approach.

Definition 1. *Fix a Markov chain* $\mathsf{M} = (S, \delta)$ *and initial state* $\hat{s}$. *Let* $\pi^{n}_{\mathsf{M},\hat{s}}(s) \coloneqq \Pr_{\mathsf{M},\hat{s}}[\Diamond^{=n}\{s\}]$ *the probability that the system is at state* s *in step* n. *Then,* $\pi^{\infty}_{\mathsf{M},\hat{s}}(s) \coloneqq \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} \pi^{i}_{\mathsf{M},\hat{s}}(s)$ *is the stationary distribution of* M.

See Fig. 1 for an example. Whenever the reference is clear from context, we omit the respective subscripts from $\pi^{\infty}_{\mathsf{M},\hat{s}}$.

We briefly recall the classical approach to compute stationary distributions (see e.g. [19, Sec. 4.7]). By Lemma 1, almost all runs eventually end up in a BSCC. Thus, $\pi^{\infty}(s) = 0$ for all states s not in a BSCC, or, dually, $\sum_{s\in B} \pi^{\infty}(s) = 1$ for $B = \bigcup_{R\in\mathrm{BSCC}(\mathsf{M})} R$. Moreover, once in a BSCC, we always obtain the same stationary distribution, irrespective of through which state we entered the BSCC. Formally, for each BSCC $R \in \mathrm{BSCC}(\mathsf{M})$ and $s, s' \in R$, we have that $\pi^{\infty}_{\mathsf{M},s} = \pi^{\infty}_{\mathsf{M},s'} = \pi^{\infty}_{\mathsf{M}|_R,s}$, i.e. each BSCC R has a unique stationary distribution, which we denote by π^{∞}_{R}. Note that $\mathrm{supp}(\pi^{\infty}_{R}) = R$, i.e. $\pi^{\infty}_{R}(s) \neq 0$ if and only if $s \in R$. Together, we observe that the stationary distribution of a Markov chain decomposes into (i) the steady state distribution in each BSCC and (ii) the probability to end up in a particular BSCC. More formally, for any state $s \in S$

$$\pi^{\infty}_{\mathsf{M},\hat{s}}(s) = \sum\nolimits_{R\in\mathrm{BSCC}(\mathsf{M})} \Pr_{\mathsf{M},\hat{s}}[\Diamond R] \cdot \pi^{\infty}_{R}(s). \tag{3}$$

Consider the example of Fig. 1: We have two BSCCs, $\{p\}$ and $\{q_1, q_2\}$, which both are reached with probability $\frac{1}{2}$, respectively. The overall distribution $\pi^{\infty}_{\mathsf{M},s}$ then is obtained from $\pi^{\infty}_{\{p\}} = \{p \mapsto 1\}$ and $\pi^{\infty}_{\{q_1,q_2\}} = \{q_1 \mapsto \frac{1}{6}, q_2 \mapsto \frac{5}{6}\}$.

As mentioned, we can compute reachability probabilities in Markov chains by solving Eq. (1). Thus, the remaining concern is to compute π^{∞}_{R}, i.e. the stationary distribution of $\mathsf{M}|_R$. In this case, i.e. Markov chains comprising a single BSCC, the steady state distribution is the unique fixed point of the transition function (up to rescaling). By defining the row transition matrix of M as $P_{i,j} = \delta(i,j)$, we can reformulate this property in terms of linear algebra. In particular, we have that $P \cdot \pi^{\infty}_{R} = \pi^{\infty}_{R}$, or, in other words, $(P - I) \cdot \pi^{\infty}_{R} = \vec{0}$, where I is an appropriately sized identity matrix [29, Thm. A.2]. This equation again can be solved by classical methods from linear algebra. In summary, we (i) compute $\mathrm{BSCC}(\mathsf{M})$, (ii) for each BSCC R, compute π^{∞}_{R} and $\Pr_{\mathsf{M},\hat{s}}[\Diamond R]$, and (iii) combine according to Eq. (3).

However, as also mentioned in the introduction, precisely solving linear equation systems may not scale well, both due to time as well as memory constraints. Thus, we also are interested in relaxing the problem slightly and instead *approximating* the stationary distribution up to a given precision of $\varepsilon > 0$.

Problem Statement Given a Markov chain M and precision requirement $\varepsilon > 0$, compute bounds $l, u : S \to [0,1]$ such that (i) $\max_{s \in S} u(s) - l(s) \leq \varepsilon$ and (ii) for all $s \in S$ we have $l(s) \leq \pi^{\infty}_{\mathsf{M},s}(s) \leq u(s)$.

Approximate Solutions Aiming for approximations is not a new idea; to achieve practical performance, current model checkers employ approximate, iterative methods by default for most queries (typically a variant value iteration). In particular, this also is the case for stationary distribution: Instead of solving the equation system for each BSCC R precisely, we can approximate the solution by, e.g., the *power method*. This essentially means to repeatedly apply the transition matrix (of the model restricted to the BSCC) to an initial vector v_0, i.e. iterating $v_{n+1} = P_R \cdot v_n$ (or $v_{n+1} = P_R^n \cdot v_1$). Similarly, the reachability probability for each BSCC then also is approximated by value iteration.

It is known that (for aperiodic MC) $\lim_{n\to\infty} v_n = \pi^{\infty}_R$ (see e.g. [31,16,27]), however convergence up to a precision of ε may take exponential time in the worst case. Moreover, there is no known stopping criterion which allows us to detect that we have converged and stop the computation early. Yet, similar to reachability [7,14], current model checkers employ this method without a sound stopping criterion, leading to potentially arbitrarily wrong results, as we show in our evaluation (Fig. 2). See [16] for a related, in-depth discussion of these issues in the context of CTMC.

We thus want to find efficient methods to derive safe bounds on the stationary distribution of a BSCC with a correct stopping criterion and combine it with correct reachability approximations to obtain an overall fast and sound approximation. To this end, we exploit two further concepts.

Partial Exploration Recent works [7,2,18,24] demonstrate the applicability of *partial exploration* to a variety of problems associated with probabilistic systems such as reachability. Essentially, the idea is to "omit" parts of the system which can be proven to be irrelevant for the result, instead focussing on important areas of the system. Of course, by omitting parts of the system, we may incur a small error. As such, these approaches naturally aim for approximate solutions.

Mean payoff We make use of another property, namely *mean payoff* (also known as *long-run average reward*). We provide a brief overview and direct to e.g. [29, Chp. 8 & 9] or [2] for more information. Mean payoff is specified by a Markov chain and a *reward function* $r : S \to \mathbb{R}$, assigning a reward to each state. Given an infinite path $\rho = s_1 s_2 \cdots$, this naturally induces a stream of rewards $r(\rho) \coloneqq r(s_1)r(s_2)\cdots$. The mean payoff of this path then equals the average reward obtained in the limit, $\mathrm{mp}'_r(\rho) \coloneqq \liminf_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} r(s_i)$. (The limit

might not be defined for some paths, hence considering the lim inf is necessary.) Finally, the mean payoff of a state s is the *expected mean payoff* according to $\mathsf{Pr}_{\mathsf{M},s}$, i.e. $\mathrm{mp}_r(s) \coloneqq \mathbb{E}_{\mathsf{M},s}[\mathrm{mp}'_r]$.

Classically, mean payoff is computed by solving a linear equation system [29, Thm. 9.1.2]. Instead, we can also employ value iteration to approximate the mean payoff, however with a slight twist. We iteratively compute the *expected total reward*, i.e. the expected sum of rewards obtained after n steps, by iterating $v_{n+1}(s) = r(s) + \delta(s)\langle v_n \rangle$. It turns out that the *increase* $\Delta_n(s) = v_{n+1}(s) - v_n(s)$ approximates the mean payoff, i.e. $\mathrm{mp}_r(s) = \lim_{n\to\infty} \Delta_n(s)$ [29, Thm. 9.4.5 a)]. Moreover, we have $\min_{s'\in S} \Delta_n(s') \leq \mathrm{mp}_r(s) \leq \max_{s'\in S} \Delta_n(s')$, yielding a correct stopping criterion [29, Thm. 9.4.5 b)]. Finally, on BSCCs these upper and lower bounds always converge [29, Cor. 9.4.6 b)], yielding termination guarantees. We provide further details on VI for mean payoff in [26, App. A.3].

3 Building Blocks

To arrive at a practical algorithm approximating the stationary distribution, we propose to employ sampling-based techniques, inspired by, e.g. [7,2,18]. Intuitively, these approaches repeatedly sample paths and compute bounds on a single property such as reachability or mean payoff. The sampling is designed to follow probable paths with high probability, hence the computation automatically focuses on the most relevant parts of the system. Additionally, by building the system *on the fly*, construction of hardly reachable parts of the system may be avoided altogether, yielding immense speed-ups for some models (see, e.g., [18] for additional background). We apply a series of tweaks to the original idea to tailor this approach to our use case, i.e. approximating the stationary distribution.

In this section, we present the "building blocks" for our approximate approach. In the spirit of Eq. (3), we discuss how we handle a single BSCC and how to approximate the reachability probabilities of all BSCCs. In the following section, we then combine these two approaches in a non-trivial manner.

3.1 Bounds in BSSCs through Mean Payoff

It is well known that the mean payoff can be computed directly from the stationary distribution [29, Prop. 8.1.1], namely:

$$\mathrm{mp}_r(s) = \sum\nolimits_{s'\in S} \pi^{\infty}_{\mathsf{M},s}(s') \cdot r(s') \tag{4}$$

In this section, we propose the opposite, namely computing the stationary distribution of a BSCC through mean payoff queries. Fix a Markov chain $\mathsf{M} = (S, \delta)$ which comprises a single BSCC, i.e. $S \in \mathrm{BSCC}(\mathsf{M})$, and define $r(s') = \mathbb{1}_{\{s\}}(s')$, i.e. 1 for s and 0 otherwise. Then, the mean payoff corresponds to the frequency of s appearing, i.e. the stationary distribution. Formally, we have that $\pi^{\infty}_{\mathsf{M},\hat{s}}(s) = \mathrm{mp}_r(s')$ for any state s' (in a BSCC, all states have the same value). This also follows directly by inserting in Eq. (4). So, naively, for each state of the BSCC, we can solve a mean payoff query, and from these results obtain the overall stationary distribution.

Algorithm 1 Approximate Stationary Distribution in BSCC

Input: Markov chain $\mathsf{M} = (S, \delta)$ with $\mathrm{BSCC}(\mathsf{M}) = \{S\}$
Output: Bounds l, u on stationary distribution π_S^∞.

$n \leftarrow 1$
for $s \in S$ **do** $l_1(s) \leftarrow 0$, $u_1(s) \leftarrow 1$
for $s \in S$ **do**
  $m \leftarrow 1$, $v_1 \leftarrow \textsc{InitGuess}(s)$
  while not $\textsc{ShouldStop}(s, m, \Delta_m)$ **do** ▷ *Iterate until some stopping criterion*
    for $s' \in S$ **do** $v_{m+1}(s') \leftarrow \mathbb{1}_{\{s\}}(s') + \delta(s')\langle v_m \rangle$ ▷ *Mean payoff VI for s*
    $m \leftarrow m + 1$
  $l'_n(s) \leftarrow \max\left(l_n(s), \min_{s' \in S} \Delta_m(s')\right)$, $u'_n(s) \leftarrow \min\left(u_n(s), \max_{s' \in S} \Delta_m(s')\right)$
  for $s' \in S \setminus \{s\}$ **do** $l'_n(s') \leftarrow l_n(s')$, $u'_n(s') \leftarrow u_n(s')$
  for $s' \in S$ **do** ▷ *Update bounds based on current results (optional)*
    $l_{n+1}(s') \leftarrow \max\left(l'_n(s'), 1 - \sum_{s'' \in S, s'' \neq s'} u'_n(s'')\right)$
    $u_{n+1}(s') \leftarrow \min\left(u'_n(s'), 1 - \sum_{s'' \in S, s'' \neq s'} l'_n(s'')\right)$
  $n \leftarrow n + 1$ and copy all unchanged values from n to $n + 1$
return (l_n, u_n)

At first, this may seem excessive, especially considering that computing the complete stationary distribution is as hard as determining the mean payoff for one state (both can be obtained by solving a linearly sized equation system). However, this idea yields some interesting benefits. Firstly, using the approximation approach discussed in Section 2, we obtain a practical approximation scheme with converging bounds for each state. As such, we can quickly stop the computation if the bounds converge fast. Moreover, we can pause and restart the computation for each state, which we will use later on in order to focus on crucial states. Finally, observe that π_R^∞ is a distribution. Thus, having lower bounds on some states actually already yields upper bounds for remaining states. Formally, for some lower bound $l : S \to [0, 1]$, we have $\pi_R^\infty(s) \leq 1 - \sum_{s' \in S, s' \neq s} l(s')$. If during our computation it turns out that a few states are actually visited very frequently, i.e. the sum of their lower bounds is close to 1, we can already stop the computation without ever investigating the other states. Note that this only is possible since we obtain provably correct bounds.

Combining these ideas, we present our first algorithm template in Algorithm 1. We solve each state separately, by applying the classical value iteration approach for mean payoff until a termination criterion is satisfied. To allow for modifications, we leave the definition of several sub-procedures open. Firstly, InitGuess initializes the value vector for each mean payoff computation. We can naively choose 0 everywhere, obtain an initial guess by heuristics, or re-use previously computed values. Secondly, ShouldStop decides when to stop the iteration for each state. A simple choice is to iterate until $\max \Delta_m(s) - \min \Delta_m(s) < \varepsilon$ for some precision requirement ε. By results on mean payoff, we can conclude that in this case the stationary distribution is computed with a precision of ε. However, as we argue later on, more sophisticated choices are possible. Finally, the order in which states are chosen is not fixed. Indeed, any order yields correct results, however heuristically re-ordering the states may also bring practical benefits.

Before we continue, we briefly argue that the algorithm is correct.

Theorem 1. *The result returned by Algorithm 1 is correct for any MC* $\mathsf{M} = (S, \delta)$ *with* $\mathrm{BSCC}(\mathsf{M}) = \{S\}$.

Proof (Sketch). Correctness of the mean payoff iteration follows from the definition of the reward function, Eq. (4), and the correctness of value iteration for mean payoff [29, Sec. 8.5]. In particular, note that the states of the MC form a single BSCC and the model is *unichain* (see [29, Chp. A]), implying that all states have the same value. For l and u, we prove correctness inductively. The initial values are trivially correct. The updates based on the mean payoff computation are correct by the above arguments and by induction hypothesis: The maximum of two correct lower bounds still is a lower bound, analogous for the upper bound. The updates based on the bounds are correct since π_R^∞ is a distribution and l', u' are correct bounds. □

We deliberately omit introducing an explicit precision requirement in the algorithm, since we will use it as a building block later on.

Remark 1. A variant of this approach also allows for memory savings: By handling one state at a time, we only need to store linearly many additional values (in the number of states) at any time, while an explicit equation system may require quadratic space. This only yields a constant factor improvement if the system is represented explicitly (storing δ requires as much space), however can be of significant merit for symbolically encoded systems. Note that this comes at a cost: As we cannot stop and resume the computation for different states, we have to determine the correct result up to the required precision immediately.

3.2 Reachability and Guided Sampling

As mentioned before, the second challenge to obtain a stationary distribution is the reachability probability for each BSCC. We employ a sampling-based approach using insights from [7]. There, the authors considered a single reachability objective, i.e. a single value per state. In contrast, we need to bound reachability probabilities for each BSCC. For now, suppose that all BSCCs are already discovered and their respective stationary distribution is already computed (or approximated). In other words, we have for each BSCC $R \in \mathrm{BSCC}(\mathsf{M})$ bounds $l^R, u^R : R \to [0,1]$ with $l_R(s) \leq \pi_R^\infty(s) \leq u_R(s)$, and we want to obtain bounds on the stationary distribution, i.e. functions l, u such that $l(s) \leq \pi_{\mathsf{M},\hat{s}}^\infty(s) \leq u(s)$. We propose to additionally compute bounds on the probability to reach each BSCC R, i.e. functions $l^{\Diamond R}$ and $u^{\Diamond R}$ such that $l^{\Diamond R}(s) \leq \mathsf{Pr}_{\mathsf{M},s}[\Diamond R] \leq u^{\Diamond R}(s)$. By Eq. (3), we then have for each state s a bound on the stationary distribution

$$\sum\nolimits_{R \in \mathrm{BSCC}(\mathsf{M})} l^{\Diamond R}(\hat{s}) \cdot l^R(s) \leq \pi_{\mathsf{M},\hat{s}}^\infty(s) \leq \sum\nolimits_{R \in \mathrm{BSCC}(\mathsf{M})} u^{\Diamond R}(\hat{s}) \cdot u^R(s).$$

We take a route similar to [7]. There, the algorithm essentially samples a path through the system, possibly guided by a heuristic, terminates the sampling based on several criteria, and then propagates the reachability value backwards along the path, repeating until termination. We propose a simple modification, namely to sample until a BSCC is reached, and then propagate the reachability

Algorithm 2 Approximate BSCC Reachability

Input: Markov chain $\mathsf{M} = (S, \delta)$
Output: For each BSCC R bounds $l^{\Diamond R}, u^{\Diamond R}$ on the probability to reach R.

$B \leftarrow \bigcup_{R \in \mathrm{BSCC}(\mathsf{M})} R$, $n \leftarrow 1$
for $R \in \mathrm{BSCC}(\mathsf{M})$ **do**
  for $s \in R$ **do** $l_1^{\Diamond R}(s) \leftarrow 1$, $u_1^{\Diamond R}(s) \leftarrow 1$
  for $s \in B \setminus R$ **do** $l_1^{\Diamond R}(s) \leftarrow 0$, $u_1^{\Diamond R}(s) \leftarrow 0$
  for $s \in S \setminus B$ **do** $l_1^{\Diamond R}(s) \leftarrow 0$, $u_1^{\Diamond R}(s) \leftarrow 1$
while ShouldSample **do** ▷ *Sample until some stopping criterion*
  $P \leftarrow$ SampleStates ▷ *Select states to update (e.g. sample a path)*
  for $R \in$ SelectUpdate(P) **do** ▷ *Select BSCCs to update*
    for $s \in P$ **do**
      $l_{n+1}^{\Diamond R}(s) \leftarrow \delta(s)\langle l_n^{\Diamond R} \rangle$
      $u_{n+1}^{\Diamond R}(s) \leftarrow \delta(s)\langle u_n^{\Diamond R} \rangle$
  for $s \in S$ **do** ▷ *Update bounds based on current results (optional)*
    for $R \in \mathrm{BSCC}(\mathsf{M})$ **do**
      $l_{n+1}^{\Diamond R}(s) \leftarrow \max\left(l_n^{\Diamond R}(s), 1 - \sum_{R' \in \mathrm{BSCC}(\mathsf{M}), R' \neq R} u_n^{R'}(s)\right)$
      $u_{n+1}^{\Diamond R}(s) \leftarrow \min\left(u_n^{\Diamond R}(s), 1 - \sum_{R' \in \mathrm{BSCC}(\mathsf{M}), R' \neq R} l_n^{R'}(s)\right)$
  $n \leftarrow n + 1$ and copy unchanged values from $l_n^{\Diamond R}$ and $u_n^{\Diamond R}$ to $l_{n+1}^{\Diamond R}$ and $u_{n+1}^{\Diamond R}$
return $\{(l^{\Diamond R}, u^{\Diamond R}) \mid R \in \mathrm{BSCC}(R)\}$

values of that particular BSCC back along the path. Moreover, we can employ a similar trick as above: Due to Lemma 1, the reachability probabilities of BSCCs sum up to one, i.e. $\sum_{R \in \mathrm{BSCC}(\mathsf{M})} \Pr_{\mathsf{M},s}[\Diamond R] = 1$ for every state s. Hence, the sum of lower bounds also yields upper bounds for other BSCCs, even those we have never encountered so far.

Our ideas are summarized in Algorithm 2. As before, the algorithm leaves several choices open. Instead of requiring to sample a path, our algorithm allows to select an arbitrary set of states to update. We note that the exact choice of this sampling mechanism does not improve the worst case runtime. However, as first observed in [7], specially crafted *guidance heuristics* can achieve dramatic practical speed-ups on several models. Later on, we combine our two algorithms and derive such a heuristic. For now, we briefly prove correctness.

Theorem 2. *The result returned by Algorithm 2 is correct for any MC* $\mathsf{M} = (S, \delta)$ *with* $\mathrm{BSCC}(\mathsf{M}) = \{S\}$.

Proof (Sketch). Similar to the previous algorithm, we prove correctness by induction. The initial values for $l^{\Diamond R}$ and $u^{\Diamond R}$ are correct. Then, assume that $l_n^{\Diamond R}$ and $u_n^{\Diamond R}$ are correct bounds. The correctness of the back propagation updates follows directly by inserting in Eq. (1) (or other works on interval value iteration [7,14]). Updates based on the bounds in other states are correct by Lemma 1 – the sum of all BSCC reachability probabilities is 1. Together, this yields correctness of the bounds computed by the algorithm. □

To obtain termination, it is sufficient to require that every state eventually is selected "arbitrarily often" by SampleStates. However, as before, we delegate the termination proof to our combined algorithm in the following section.

4 Dynamic Computation with Partial Exploration

Recall that our overarching goal is to approximate the stationary distribution through Eq. (4). In the previous section, we have seen how we can (i) obtain approximations for a given BSCC and (ii) how to approximate the reachability probabilities of all BSCCs through sampling. However, the naive combination of these algorithms would require us to compute the set of all BSCCs, approximate the stationary distribution in each of them until a fixed precision, and additionally approximate reachability for each of them.

We now combine both ideas to obtain a sampling-based algorithm, capable of partial exploration, that focusses computation on relevant parts of the system. In particular, we construct the system dynamically, identify BSCCs on the fly, and interleave the exploration with both the approximation inside each explored BSCC (Algorithm 1) and the overall reachability computation (Algorithm 2). Moreover, we focus computation on BSCCs which are likely to be reached and thus have a higher impact on the overall error of the result. Together, our approach roughly performs the following steps until the required precision is achieved:

- Sample a path through the system, guided by a heuristic,
- check if a new BSCCs is discovered or sampling ended in a known BSCC,
- refine bounds on the stationary distribution in the reached BSCC, and
- propagate reachability bounds and additional information along the path.

We first formalize a generic framework which can instantiate the classical, precise approach as well as our approximation building blocks and then explain our concrete variant of this framework to efficiently obtain ε-precise bounds.

4.1 The Framework

Since our goal is to allow for both precise as well as approximate solutions, we phrase the framework using lower and upper bounds together with abstract refinement procedures. We first explain our algorithm and how it generalizes the classical approach. Then, we prove its correctness under general assumptions. Finally, we discuss several approximate variants.

Algorithm 3 essentially repeats three steps until the termination condition in Line 4 is satisfied. First, we update the set of known BSCCs through UpdateBSSCs. In the classical solution, this function simply computes BSCC(M) once; our on-the-fly construction would repeatedly check for newly discovered BSCCs, dynamically growing the set $\mathcal{B}_n$. Then, we select BSCCs for which we should update the stationary distribution bounds. The classical solution solves the fixed point equation we have discussed in Section 2 for all BSCCs, i.e. SelectDistributionUpdates yields BSCC(M) and RefineDistribution the precisely computed values both as upper and lower bounds. Alternatively, we could, for example, select a single BSCC and apply a few iterations of Algorithm 1. Next, we update reachability bounds for a selected set of BSCCs. Again, the classical solution solves the reachability problem precisely for each BSCC through Eq. (1). Instead, we could employ value iteration as suggested by Algorithm 2.

Algorithm 3 Stationary Distribution Computation Framework

Input: Markov chain $\mathsf{M} = (S, \delta)$, initial state $\hat{s}$, precision $\varepsilon > 0$
Output: ε-precise bounds l, u on the stationary distribution $\pi^{\infty}_{\mathsf{M},\hat{s}}$

for $s \in S$ **do** ▷ *Initial bounds for all possible BSSCs that can be discovered*
  $l_1^{\Diamond\circ}(s) = 0$, $u_1^{\Diamond\circ}(s) = 1$, $l_1^{\circ}(s) \leftarrow 0$, $u_1^{\circ}(s) \leftarrow 1$
$n \leftarrow 1$, $\mathcal{B}_1 \leftarrow \emptyset$
while $\left(1 - \sum_{R \in \mathcal{B}_n} l_n^{\Diamond R}(\hat{s})\right) + \sum_{R \in \mathcal{B}_n} \left(l_n^{\Diamond R}(\hat{s}) \cdot \max_{s \in S}(u_n^R(s) - l_n^R(s))\right) > \varepsilon$ **do**
  $n \leftarrow n + 1$
  $\mathcal{B}_n \leftarrow$ UpdateBSSCs, $B_n \leftarrow \bigcup_{R \in \mathcal{B}_n} R$ ▷ *Discover new BSCCs*
  for $R \in \mathcal{B}_n \setminus \mathcal{B}_{n-1}$, $s \in R$ **do** ▷ *Update trivial reach bounds*
    $l_n^{\Diamond R}(s) \leftarrow 1$ ▷ *$s \in R$ surely reaches R*
    for $\circ \neq R$ **do** $u_n^{\Diamond\circ}(s) \leftarrow 0$ ▷ *$s \in R$ reaches no other BSCC*
  for $R \in$ SelectDistributionUpdates$(\mathcal{B}_n) \cap \mathcal{B}_n$ **do**
    $(l_n^R, u_n^R) \leftarrow$ RefineDistribution(R) ▷ *Update BSCC bounds*
  for $R \in$ SelectReachUpdates$(\mathcal{B}_n) \cap \mathcal{B}_n$ **do**
    $(l_n^{\Diamond R}, u_n^{\Diamond R}) \leftarrow$ RefineReach(R) ▷ *Update reachability bounds*
  Copy unchanged variables from $n-1$ to n
$L \leftarrow \sum_{R \in \mathcal{B}_n} l_n^{\Diamond R}(\hat{s})$
for $R \in \mathcal{B}_n$, $s \in R$ **do**
  $l(s) \leftarrow l_n^{\Diamond R}(\hat{s}) \cdot l_n^R(s)$
  $u(s) \leftarrow \min(u_n^{\Diamond R}(\hat{s}), 1 - L + l_n^{\Diamond R}(\hat{s})) \cdot u_n^R(s)$
for $s \in S \setminus B_n$ **do** $l(s) \leftarrow 0$, $u(s) \leftarrow 0$
return (l, u)

Before we present our variant, we prove correctness under weak assumptions. We note a subtlety of the termination condition: One may assume that upper bounds on the reachability are required to bound the overall error caused by each BSCC. Yet, as we show in the following theorem, *lower* bounds are sufficient. The upper bound is implicitly handled by the first part of the termination condition.

Theorem 3. *The result returned by Algorithm 3 is correct, i.e. ε precise bounds on the stationary distribution, if (i) $\mathcal{B}_n \subseteq \mathcal{B}_{n+1} \subseteq \mathrm{BSCC}(\mathsf{M})$ for all n, and (ii)* RefineDistribution *and* RefineReach *yield correct, monotone bounds.*

The proof can be found in [26, App. B.1].

Remark 2. Technically, the algorithm does not need to track explicit upper bounds on the reachability of each BSCC at all. Indeed, for a BSCC $R \in \mathcal{B}_n$, we could use $1 - \sum_{R' \in \mathrm{BSCC}(\mathsf{M}) \setminus \{R\}} l_n^{\Diamond R'}(s)$ as upper bound and still obtain a correct algorithm. However, tracking a separate upper bound is easier to understand and has some practical benefits for the implementation.

We exclude a proof of termination, since this strongly depends on the interplay between the functions left open. We provide a general, technical criterion together with a proof in [26, App. B.2]. Intuitively, as one might expect, we require that eventually UpdateBSSCs identifies all relevant BSCCs, SelectDistributionUpdates and SelectReachUpdates select all relevant BSCCs, and RefineDistribution and RefineReach converge to the respective true value. In the following, we present a concrete template which satisfies this criterion.

4.2 Sampling-Based Computation

We present our instantiation of Algorithm 3 using guided sampling and heuristics. Since the details of the sampling guidance heuristic are rather technical, we focus on how the template functions UpdateBSSCs, SelectDistributionUpdates, RefineDistribution, SelectReachUpdates, and RefineReach are instantiated. For now, the reader may assume that states are, e.g., selected by sampling random paths through the system.

- UpdateBSSCs: We track the set of *explored* states, i.e. states which have already been sampled at least once. On these, we search for BSCCs whenever we repeatedly stop sampling due to a state re-appearing.
- SelectDistributionUpdates: If we stopped sampling due to entering a known BSCC, we update the bounds of this single one, otherwise none.
- RefineDistribution: We employ Algorithm 1 to refine the bounds until the error over all states is halved.
- SelectReachUpdates: We refine the reach values for all sampled states.
- RefineReach: If we stopped sampling due to entering a BSCC, we back-propagate the reachability bounds for this BSCC in the spirit of Algorithm 2, i.e. for all sampled states set $l_{n+1}^{\Diamond R}(s) = \delta(s)\langle l_n^{\Diamond R}\rangle$ and $u_{n+1}^{\Diamond R}(s) = \delta(s)\langle u_n^{\Diamond R}\rangle$.

We prove that this yields correct results and terminates with probability 1 through Theorem 3. Note that this description leaves exact details of the sampling open. Thus, we prove termination using (weak) conditions on the sampling mechanism. For readability, we define the shorthand $\mathsf{err}_n^R = \max_{s\in R} u_n^R(s) - l_n^R(s)$ denoting the overall error of the stationary distribution in BSCC R and $\mathsf{err}_n^{\Diamond R}(s) = u_n^{\Diamond R}(s) - l_n^{\Diamond R}(s)$ the error bound on the reachability of R from s.

Theorem 4. *Algorithm 3 instantiated with our sampling-based approach yields correct results and terminates with probability 1 if, with probability 1,*

(S.i) *the sampled states $P \subseteq S$ satisfy $\mathsf{Pr}_{\mathsf{M},\hat{s}}[\Diamond\overline{P}] < \frac{\varepsilon}{4}$ (P is a $\frac{\varepsilon}{4}$-core [18]),*
(S.ii) *the initial state is sampled arbitrarily often, and*
(S.iii) *for each state s sampled arbitrarily often, every successor $s' \in P$ with $E_n(s') := \max_{R\in\mathcal{B}_n} u_n^{\Diamond R}(s') \cdot \mathsf{err}_n^R + \max_{R\in\mathcal{B}_n} \mathsf{err}_n^{\Diamond R}(s) \geq \frac{\varepsilon}{4(|\mathcal{B}_n|+1)}$ is sampled arbitrarily often,*

where "arbitrarily often" means that if the algorithm would not terminate, this would happen infinitely often.

The proof can be found in [26, App. B.3].

Due to space constraints, we omit an in-depth description of our sampling method and only provide a brief summary here. In summary, our algorithm first selects a "sampling target" which is either "the unknown", i.e. states not seen so far, to encourage exploration in the style of [18], or a known BSCC, to bias sampling towards it. We select a choice randomly, weighted by its current potential influence on the precision. The sampling process is guided by the chosen target, taking actions which lead to the respective target with high probability. In technical terms, we sample successors weighted by the upper

bound on reachability probability times the transition probability. Once the target is reached, we either explore the unknown, or improve precision in the reached BSCC. Finally, information is back-propagated along the path. Further details, in particular pitfalls we encountered during the design process, together with a complete instantiation of our algorithm can be found in [26, App. C].

5 Experimental Evaluation

In this section, we evaluate our approaches, comparing to both our own reference implementation using classical methods, as well as the established model checker PRISM [21]. (The other popular model checkers Storm [10] and IscasMC/ePMC [15] do not directly support computing stationary distributions.) We implemented our methods in Java based on PET [24], running on consumer hardware (AMD Ryzen 5 3600). To solve arising linear equation systems, we use `Jeigen v1.2`. All executions are performed in a Docker container, restricted to a single CPU core and 8GB of RAM. For approximations, we require a precision of $\varepsilon = 10^{-4}$.

Tools Aside from `PRISM`[1], we consider three variants of Algorithm 3, namely `Classic`, the classical approach, solving each BSCC through a linear equation system and then approximating the reachability through PRISM (using interval iteration), `Naive`, the naive sampling approach, following the transition dynamics, and `Sample`, our sampling approach, selecting a target and steering towards it. The sourcecode of our implementation used to run these experiments as well as all models and our data is available at [25]. Moreover, the current version can be found at GitHub [23].

We mention two points relevant for the comparison. First, as we show in the following, `PRISM` may yield wrong results due to a (too) simple computation. As such, we should not expect that our correct methods are on par or even faster. Second, our implementation employs conservative procedures to further increase quality of the result, such as compensated summation to mitigate numerical error due to floating-point imprecision, noticeably increasing computational effort.

Models We consider the PRISM benchmark suite[2] [22], comprising several probabilistic models, in particular DTMC, CTMC, and MDP. Since there are not too many Markov chains in this set, we obtain further models as follows. For each CTMC, we consider the *uniformized CTMC* (which preserves the steady state distribution), and for MDP we choose actions uniformly at random. Unfortunately, *all* models obtained this way either comprise only single-state BSCCs or the whole model is a single BSCC. In the former case, our approximation within the BSCC is not used at all, in the latter, a sampling based approach needs to invest additional time to discover the whole system. In order to better compare the performance of our mean payoff based approximation approach, in these cases

[1] We observed that the default hybrid engine typically is significantly slower than the "explicit" variant and thus use that one, see [26, App. D].

[2] Obtained from `https://github.com/prismmodelchecker/prism-benchmarks`.

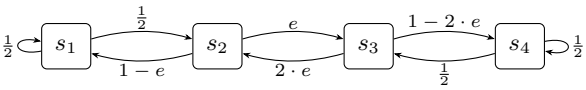


Fig. 2. A small MC where PRISM reports wrong results for $e \leq 10^{-7}$.

we pre-explore the whole system and compute the stationary distribution directly through Algorithm 1. To compare the combined performance, we additionally consider a handcrafted model, named **branch**, which comprises both transient states as well as several non-trivial BSCCs.

We present selected results, highlighting different strengths and weaknesses of each approach. An evaluation of the complete suite can be found in [26, App. D].

Correctness We discovered that `PRISM` potentially yields wrong results, due to an unsafe stopping criterion. In particular, `PRISM` iterates the power method until the absolute difference between subsequent iterates is small, exactly as with its "unsafe" value iteration for reachability, as reported by e.g. [7]. On the model from Fig. 2, PRISM (with explicit engine) immediately terminates, printing a result of $\approx (\frac{1}{6}, \frac{1}{6}, \frac{1}{3}, \frac{1}{3})$. However, the correct stationary distribution is $\approx (\frac{1}{9}, \frac{2}{9}, \frac{4}{9}, \frac{2}{9})$ (from left to right), which both of our methods correctly identify. This behaviour is due to the small difference between first and second eigenvalue of the transition matrix, which in turn implies that the iterates of the power method only change by a small amount. We note that on this example, PRISM's default hybrid engine eventually yields the correct result (after $\approx 10^8$ iterations) due to the used iteration scheme. On small variation of the model (included in the artefact) it also terminates immediately with the wrong result.

Results We summarize our results in Table 1. We observe several points. First, we see that the naive sampling approach can hardly handle non-trivial models. Second, our guided sampling approach achieves significant improvements on several models over both the classical, correct method as well as the potentially unsound approach of PRISM, in particular when hardly reachable portions of the state space can be completely discarded. However, on other models, the classical approach seems to be more appropriate, in particular on models with many likely to be reached BSCCs. Here, the sampling approach struggles to propagate the reachability bounds of all BSCCs simultaneously. Finally, as suggested by the **phil** and **rabin** models, using mean payoff based approximation can significantly outperform classical equation solving. In summary, `PRISM`, `Classic`, and `Sample` all can be the fastest method, depending on the structure of the model. However, recall that `PRISM`'s method does not give guarantees on the result.

Further Discussion As expected, we observed that the runtime of approximation can increase drastically for smaller precision requirements (e.g. $\varepsilon = 10^{-8}$) and solving the equation system precisely may actually be faster for some BSCCs. However, especially in the combined approach, if we already have some upper bounds on the reachability probability of a certain BSCC, we do not need to solve it with the original precision. Hence, a future version of the implementation could

Table 1. Overview of our results. For each model, we list its parameters, overall size, and number of BSCCs, followed by the total execution time in seconds for each tool, TO denotes a timeout (300 seconds), MO a memout, and `err` an internal error. On systems comprising a single BSCC, the `Naive` and `Sample` approach coincide.

Model	Parameters	$\|S\|$	|BSCC|	`PRISM`	`Classic`	`Naive`	`Sample`
brp	`N=64,MAX=5`	5,192	134	1.2	11	TO	4.9
nand	`N=15,K=2`	56,128	16	4.9	30	TO	64
zeroconf_dl	`reset=false,deadline=40,N=1000,K=1`	251,740	10,048	99	238	8.0	1.0
phil4		9,440	1	`err`	TO	51	
rabin3		27,766	1	`err`	MO	178	
branch		1,087,079	1,000	155	TO	TO	20

dynamically decide whether to solve a BSCC based on mean payoff approximation or equation solving, combining advantages of both worlds.

Secondly, this also highlights an interesting trade-off implicit to our approach: The algorithm needs to balance between exploring unknown areas and refining bounds on known BSCCs, in particular, since exploring a new BSCC adds noticeable effort: One more target for which the reachability has to be determined. Here, more sophisticated heuristics could be useful.

Finally, for models with large BSCCs, such as **rabin**, we also observed that the classical linear equation approach indeed runs out of memory while a variant of the approximation algorithm can still solve it, as indicated by Remark 1. Thus, the implementation could moreover take memory constraints into account, deciding to apply the memory-saving approach in appropriate cases.

6 Conclusion

We presented a new perspective on computing the stationary distribution in Markov chains by rephrasing the problem in terms of mean payoff and reachability. We combined several recent advances for these problems to obtain a sophisticated partial-exploration based algorithm. Our evaluation shows that on several models our new approach is significantly more performant. As a major technical contribution, we provided a general algorithmic framework, which encompasses both the classical solution approach as well as our new method.

As hinted by the discussion above, our framework is quite flexible. For future work, we particularly want to identify better guidance heuristics. Specifically, based on experimental data, we conjecture that the reachability part can be improved significantly. Moreover, due to the flexibility of our framework, we can apply different methods for each BSCC to obtain the reachability and stationary distribution. Thus, we want to find meta-heuristics which suggest the most appropriate method in each case. For example, for smaller BSCCs, we could use the classical, precise solution method to obtain the stationary distribution, while for larger ones we employ our mean payoff approach, and, in the spirit of Remark 1, for even larger ones we approximate them to the required precision immediately, saving memory. Additionally, we could identify BSCCs that satisfy the conditions of specialized approaches such as [11].

References

1. Adan, I.J.B.F., Foley, R.D., McDonald, D.R.: Exact asymptotics for the stationary distribution of a markov chain: a production model. Queueing Syst. Theory Appl. **62**(4), 311–344 (2009). https://doi.org/10.1007/s11134-009-9140-y
2. Ashok, P., Chatterjee, K., Daca, P., Kretínský, J., Meggendorfer, T.: Value iteration for long-run average reward in markov decision processes. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10426, pp. 201–221. Springer (2017). https://doi.org/10.1007/978-3-319-63387-9_10
3. Backenköhler, M., Bortolussi, L., Großmann, G., Wolf, V.: Abstraction-guided truncations for stationary distributions of markov population models. In: Abate, A., Marin, A. (eds.) Quantitative Evaluation of Systems - 18th International Conference, QEST 2021, Paris, France, August 23-27, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12846, pp. 351–371. Springer (2021). https://doi.org/10.1007/978-3-030-85172-9_19
4. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
5. Bellman, R.: Dynamic programming. Science **153**(3731), 34–37 (1966)
6. Billingsley, P.: Probability and measure. John Wiley & Sons (2008)
7. Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Kretínský, J., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Verification of markov decision processes using learning algorithms. In: Cassez, F., Raskin, J. (eds.) Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8837, pp. 98–114. Springer (2014). https://doi.org/10.1007/978-3-319-11936-6_8
8. Bressan, M., Peserico, E., Pretto, L.: On approximating the stationary distribution of time-reversible markov chains. Theory Comput. Syst. **64**(3), 444–466 (2020). https://doi.org/10.1007/s00224-019-09921-3
9. Busic, A., Fourneau, J.: Iterative component-wise bounds for the steady-state distribution of a markov chain. Numer. Linear Algebra Appl. **18**(6), 1031–1049 (2011). https://doi.org/10.1002/nla.824
10. Dehnert, C., Junges, S., Katoen, J., Volk, M.: A storm is coming: A modern probabilistic model checker. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 592–600. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_31
11. Fourneau, J., Quessette, F.: Some improvements for the computation of the steady-state distribution of a markov chain by monotone sequences of vectors. In: Al-Begain, K., Fiems, D., Vincent, J. (eds.) Analytical and Stochastic Modeling Techniques and Applications - 19th International Conference, ASMTA 2012, Grenoble, France, June 4-6, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7314, pp. 178–192. Springer (2012). https://doi.org/10.1007/978-3-642-30782-9_13
12. Gillespie, D.T.: A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. Journal of computational physics **22**(4), 403–434 (1976)
13. Gómez, S., Arenas, A., Borge-Holthoefer, J., Meloni, S., Moreno, Y.: Discrete-time markov chain approach to contact-based disease spreading in complex networks. EPL **89**(3), 38009 (feb 2010). https://doi.org/10.1209/0295-5075/89/38009
14. 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

15. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasmc: A web-based probabilistic model checker. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8442, pp. 312–317. Springer (2014). https://doi.org/10.1007/978-3-319-06410-9_22
16. Katoen, J., Zapreev, I.S.: Safe on-the-fly steady-state detection for time-bounded reachability. In: Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA. pp. 301–310. IEEE Computer Society (2006). https://doi.org/10.1109/QEST.2006.47
17. Kimura, T., Masuyama, H.: A heavy-traffic-limit formula for the moments of the stationary distribution in GI/G/1-type markov chains. Oper. Res. Lett. **49**(6), 862–867 (2021). https://doi.org/10.1016/j.orl.2021.10.003
18. Kretínský, J., Meggendorfer, T.: Of cores: A partial-exploration framework for markov decision processes. Log. Methods Comput. Sci. **16**(4) (2020), https://lmcs.episciences.org/6833
19. Kulkarni, V.G.: Modeling and analysis of stochastic systems. CRC Press (2016)
20. Kuntz, J., Thomas, P., Stan, G., Barahona, M.: Stationary distributions of continuous-time markov chains: A review of theory and truncation-based approximations. SIAM Rev. **63**(1), 3–64 (2021). https://doi.org/10.1137/19M1289625
21. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47
22. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012. pp. 203–204. IEEE Computer Society (2012). https://doi.org/10.1109/QEST.2012.14
23. Meggendorfer, T.: Stationary distribution sampling, https://github.com/incaseoftrouble/stationary-distribution-sampling
24. Meggendorfer, T.: PET - A partial exploration tool for probabilistic verification. In: Bouajjani, A., Holík, L., Wu, Z. (eds.) Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13505, pp. 320–326. Springer (2022). https://doi.org/10.1007/978-3-031-19992-9_20, https://doi.org/10.1007/978-3-031-19992-9_20
25. Meggendorfer, T.: Artefact for: Correct Approximation of Stationary Distributions (Jan 2023). https://doi.org/10.5281/zenodo.7548215
26. Meggendorfer, T.: Correct approximation of stationary distributions. CoRR **abs/2301.08137** (2023). https://doi.org/10.48550/arXiv.2301.08137
27. Nesterov, Y.E., Nemirovski, A.: Finding the stationary states of markov chains by iterative methods. Appl. Math. Comput. **255**, 58–65 (2015). https://doi.org/10.1016/j.amc.2014.04.053
28. Paulsson, J.: Summing up the noise in gene networks. Nature **427**(6973), 415–418 (2004)
29. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley (1994). https://doi.org/10.1002/9780470316887
30. Spieler, D., Wolf, V.: Efficient steady state analysis of multimodal markov chains. In: Dudin, A.N., Turck, K.D. (eds.) Analytical and Stochastic Modelling Techniques and Applications - 20th International Conference, ASMTA 2013, Ghent, Belgium,

July 8-10, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7984, pp. 380–395. Springer (2013). https://doi.org/10.1007/978-3-642-39408-9_27
31. Stewart, W.J.: Introduction to the numerical solution of Markov Chains. Princeton University Press (1994)
32. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010


Robust Almost-Sure Reachability in Multi-Environment MDPs

Marck van der Vegt(✉), Nils Jansen, and Sebastian Junges

Radboud University, Nijmegen, The Netherlands
{marck.vandervegt,nils.jansen,sebastian.junges}@ru.nl

Abstract. Multiple-environment MDPs (MEMDPs) capture finite sets of MDPs that share the states but differ in the transition dynamics. These models form a proper subclass of partially observable MDPs (POMDPs). We consider the synthesis of policies that robustly satisfy an almost-sure reachability property in MEMDPs, that is, *one* policy that satisfies a property *for all* environments. For POMDPs, deciding the existence of robust policies is an EXPTIME-complete problem. We show that this problem is PSPACE-complete for MEMDPs, while the policies require exponential memory in general. We exploit the theoretical results to develop and implement an algorithm that shows promising results in synthesizing robust policies for various benchmarks.

1 Introduction

Markov decision processes (MDPs) are the standard formalism to model sequential decision making under uncertainty. A typical goal is to find a policy that satisfies a temporal logic specification [5]. Probabilistic model checkers such as Storm [22] and Prism [30] efficiently compute such policies. A concern, however, is the robustness against potential perturbations in the environment. MDPs cannot capture such uncertainty about the shape of the environment.

Multi-environment MDPs (MEMDPs) [36,14] contain a set of MDPs, called environments, over the same state space. The goal in MEMDPs is to find a single policy that satisfies a given specification in *all* environments. MEMDPs are, for instance, a natural model for MDPs with unknown system dynamics, where several domain experts provide their interpretation of the dynamics [11]. These different MDPs together form a MEMDP. MEMDPs also arise in other domains: The guessing of a (static) password is a natural example in security. In robotics, a MEMDP captures unknown positions of some static obstacle. One can interpret MEMDPs as a (disjoint) union of MDPs in which an agent only has partial observation, i.e., every MEMDP can be cast into a linearly larger partially observable MDP (POMDP) [27]. Indeed, some famous examples for POMDPs are in fact MEMDPs, such as *RockSample* [39] and *Hallway* [31]. Solving POMDPs is notoriously hard [32], and thus, it is worthwhile to investigate natural subclasses.

We consider *almost-sure specifications* where the probability needs to be one to reach a set of target states. In MDPs, it suffices to consider memoryless

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 508–526, 2023.
https://doi.org/10.1007/978-3-031-30823-9_26

policies. Constructing such policies can be efficiently implemented by means of a graph-search [5]. For MEMDPs, we consider the following problem:

Compute one *policy that almost-surely reaches the target in* all *environments.*

Such a policy robustly satisfies an almost-sure specification for a set of MDPs.

Our approach. Inspired by work on POMDPs, we construct a belief-observation MDP (BOMDP) [16] that tracks the states of the MDPs and the (support of the) belief over potential environments. We show that a policy satisfying the almost-sure property in the BOMDP also satisfies the property in the MEMDP.

Although the BOMDP is exponentially larger than the MEMDP, we exploit its particular structure to create a PSPACE algorithm to decide whether such a robust policy exists. The essence of the algorithm is a recursive construction of a fragment of the BOMDP, restricted to a setting in which the belief-support is fixed. Such an approach is possible, as the belief in a MEMDP behaves monotonically: Once we know that we are not in a particular environment, we never lose this knowledge. This behavior is in contrast to POMDPs, where there is no monotonic behavior in belief-supports. The difference is essential: Deciding almost-sure reachability in POMDPs is EXPTIME-complete [37,19]. In contrast, the problem of deciding whether a policy for almost-sure reachability in a MEMDP exists is indeed PSPACE-*complete*. We show the hardness using a reduction from the *true quantified Boolean formula problem*. Finally, we cannot hope to extract a policy with such an algorithm, as the smallest policy for MEMDPs may require exponential memory in the number of environments.

The PSPACE algorithm itself recomputes many results. For practical purposes, we create an algorithm that iteratively explores parts of the BOMDP. The algorithm additionally uses the MEMDP structure to generalize the set of states from which a winning policy exists and deduce efficient heuristics for guiding the exploration. The combination of these ingredients leads to an efficient and competitive prototype on top of the model checker Storm.

Related work. We categorize related work in three areas.

MEMDPs. Almost-sure reachability for MEMDPs for exactly two environments has been studied by [36]. We extend the results to arbitrarily many environments. This is nontrivial: For two environments, the decision problem has a polynomial time routine [36], whereas we show that the problem is PSPACE-complete for an arbitrary number of environments. MEMDPs and closely related models such as hidden-model MDPs, hidden-parameter MDPs, multi-model MDPs, and concurrent MDPs [11,2,40,10] have been considered for quantitative properties[1]. The typical approach is to consider approximative algorithms for the undecidable problem in POMDPs [14] or adapt reinforcement learning algorithms [3,28]. These approximations are not applicable to almost-sure properties.

POMDPs. One can build an underlying potentially infinite belief-MDP [27] that corresponds to the POMDP – using model checkers [35,7,8] to verify this MDP

[1] Hidden-parameter MDPs are different than MEMDPs in that they assume a prior over MDPs. However, for almost-sure properties, this difference is irrelevant.

can answer the question for MEMDPs. For POMDPs, almost-sure reachability is decidable in exponential time [37,19] via a construction similar to ours. Most qualitative properties beyond almost-sure reachability are undecidable [4,15]. Two dedicated algorithms that limit the search to policies with small memory requirements and employ a SAT-based approach [12,26] to this NP-hard problem [19] are implemented in Storm. We use them as baselines.

Robust models. The high-level representation of MEMDPs is structurally similar to featured MDPs [18,1] that represent sets of MDPs. The proposed techniques are called family-based model checking and compute policies for every MDP in the family, whereas we aim to find one policy for all MDPs. Interval MDPs [25,43,23] and SGs [38] do not allow for dependencies between states and thus cannot model features such as various obstacle positions. Parametric MDPs [2,44,24] assume controllable uncertainty and do not consider robustness of policies.

Contributions. We establish PSPACE-completeness for deciding almost-sure reachability in MEMDPs and show that the policies may be exponentially large. Our iterative algorithm, which is the first specific to almost-sure reachability in MEMDPs, builds fragments of the BOMDP. An empirical evaluation shows that the iterative algorithm outperforms approaches dedicated to POMDPs.

2 Problem Statement

In this section, we provide some background and formalize the problem statement.

For a set X, $\mathit{Dist}(X)$ denotes the set of probability distributions over X. For a given distribution $d \in \mathit{Dist}(X)$, we denote its support as $\mathit{Supp}(d)$. For a finite set X, let $\mathsf{unif}(X)$ denote the uniform distribution. $\mathsf{dirac}(x)$ denotes the Dirac distribution on $x \in X$. We use short-hand notation for functions and distributions, $f = [x \mapsto a, y \mapsto b]$ means that $f(x) = a$ and $f(y) = b$. We write $\mathcal{P}(X)$ for the powerset of X. For $n \in \mathbb{N}$ we write $[n] = \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$.

Definition 1 (MDP). *A* Markov Decision Process *is a tuple* $\mathcal{M} = \langle S, A, \iota_{init}, p \rangle$ *where* S *is the finite set of states,* A *is the finite set of actions,* $\iota_{init} \in \mathit{Dist}(S)$ *is the initial state distribution, and* $p\colon S \times A \to \mathit{Dist}(S)$ *is the transition function.*

The transition function is total, that is, for notational convenience MDPs are *input-enabled*. This requirement does not affect the generality of our results. A *path* of an MDP is a sequence $\pi = s_0 a_0 s_1 a_1 \dots s_n$ such that $\iota_{\text{init}}(s_0) > 0$ and $p(s_i, a_i)(s_{i+1}) > 0$ for all $0 \leq i < n$. The last state of π is $\mathit{last}(\pi) = s_n$. The set of all finite paths is Path and $\text{Path}(S')$ denotes the paths starting in a state from $S' \subseteq S$. The set of *reachable states* from S' is $\mathsf{Reachable}(S')$. If $S' = \mathit{Supp}(\iota_{\text{init}})$ we just call them *the* reachable states. The MDP restricted to reachable states from a distribution $d \in \mathit{Dist}(S)$ is $\mathsf{ReachFragment}(\mathcal{M}, d)$, where d is the new initial distribution. A state $s \in S$ is *absorbing* if $\mathsf{Reachable}(\{s\}) = \{s\}$. An MDP is acyclic, if each state is absorbing or not reachable from its successor states.

Action choices are resolved by a *policy* $\sigma\colon \text{Path} \to \mathit{Dist}(A)$ that maps paths to distributions over actions. A policy of the form $\sigma\colon S \to \mathit{Dist}(A)$ is

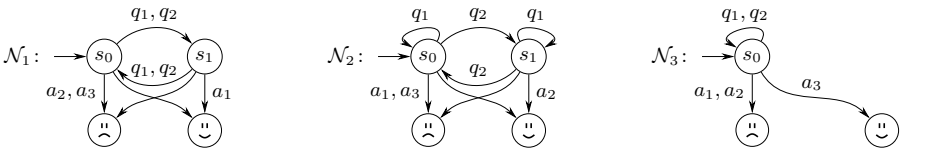


Fig. 1: Example MEMDP

called *memoryless*, *deterministic* if we have $\sigma\colon \textsc{Path} \to A$; and, *memoryless deterministic* for $\sigma\colon S \to A$. For an MDP $\mathcal{M}$, we denote the probability of a policy σ reaching some target set $T \subseteq S$ starting in state s as $\Pr_{\mathcal{M}}(s \to T \mid \sigma)$. More precisely, $\Pr_{\mathcal{M}}(s \to T \mid \sigma)$ denotes the probability of all paths from s reaching T under σ. We use $\Pr_{\mathcal{M}}(T \mid \sigma)$ if s is distributed according to ι_{init}.

Definition 2 (MEMDP). *A* Multiple Environment MDP *is a tuple* $\mathcal{N} = \langle S, A, \iota_{init}, \{p_i\}_{i\in I}\rangle$ *with* S, A, ι_{init} *as for MDPs, and* $\{p_i\}_{i\in I}$ *is a* set of transition functions*, where* I *is a finite set of* environment *indices.*

Intuitively, MEMDPs form sets of MDPs (environments) that share states and actions, but differ in the transition probabilities. For MEMDP $\mathcal{N}$ with index set I and a set $I' \subseteq I$, we define the restriction of environments as the MEMDP $\mathcal{N}_{\downarrow I'} = \langle S, A, \iota_{\text{init}}, \{p_i\}_{i\in I'}\rangle$. Given an environment $i \in I$, we denote its corresponding MDP as $\mathcal{N}_i = \langle S, A, \iota_{\text{init}}, p_i\rangle$. A MEMDP with only one environment is an MDP. Paths and policies are defined on the states and actions of MEMDPs and do not differ from MDP policies. A MEMDP is acyclic, if each MDP is acyclic.

Example 1. Figure 1 shows an MEMDP with three environments $\mathcal{N}_i$. An agent can ask two questions, q_1 and q_2. The response is either 'switch' ($s_1 \leftrightarrow s_2$), or 'stay' (loop). In $\mathcal{N}_1$, the response to q_1 and q_2 is to switch. In $\mathcal{N}_2$, the response to q_1 is stay, and to q_2 is switch. The agent can guess the environment using a_1, a_2, a_3. Guessing a_i leads to the target $\{\text{☺}\}$ only in environment i. Thus, an agent must deduce the environment via q_1, q_2 to surely reach the target. ■

Definition 3 (Almost-Sure Reachability). *An almost-sure reachability property is defined by a set* $T \subseteq S$ *of target states. A policy* σ *satisfies the property* T *for MEMDP* $\mathcal{N} = \langle S, A, \iota_{init}, \{p_i\}_{i\in I}\rangle$ *iff* $\forall i \in I\colon \Pr_{\mathcal{N}_i}(T \mid \sigma) = 1$.

In other words, a policy σ satisfies an almost-sure reachability property T, called *winning*, if and only if the probability of reaching T *within each MDP* is one. By extension, a state $s \in S$ is winning if there exists a winning policy when starting in state s. Policies and states that are not winning are losing.
We will now define both the decision and policy problem:

Given a MEMDP $\mathcal{N}$ and an almost-sure reachability property T.
The **Decision Problem** asks to decide if a policy exists that satisfies T.
The **Policy Problem** asks to compute such a policy, if it exists.

In Section 4 we discuss the computational complexity of the decision problem. Following up, in Section 5 we present our algorithm for solving the policy problem. Details on its implementation and evaluation will be presented in Section 6.

3 A Reduction To Belief-Observation MDPs

In this section, we reduce the policy problem, and thus also the decision problem, to finding a policy in an exponentially larger belief-observation MDP. This reduction is an elementary building block for the construction of our PSPACE algorithm and the practical implementation. Additional information such as proofs for statements throughout the paper are available in the technical report [41].

3.1 Interpretation of MEMDPs as Partially Observable MDPs

Definition 4 (POMDP). *A partially observable MDP (POMDP) is a tuple* $\langle \mathcal{M}, Z, O \rangle$ *with an MDP* $\mathcal{M} = \langle S, A, \iota_{init}, p \rangle$*, a set* Z *of* observations*, and an* observation function $O \colon S \to Z$*.*

A POMDP is an MDP where states are labelled with observations. We lift O to paths and use $O(\pi) = O(s_1)a_1O(s_2)\dots O(s_n)$. We use observation-based policies σ, i.e., policies s.t. for $\pi, \pi' \in \textsc{Path}$, $O(\pi) = O(\pi')$ implies $\sigma(\pi) = \sigma(\pi')$. A MEMDP can be cast into a POMDP that is made up as the disjoint union:

Definition 5 (Union-POMDP). *Given an MEMDP* $\mathcal{N} = \langle S, A, \iota_{init}, \{p_i\}_{i \in I} \rangle$ *we define its* union-POMDP $\mathcal{N}_\sqcup = \langle \langle S', A, \iota'_{init}, p' \rangle, Z, O \rangle$*, with states* $S' = S \times I$*, initial distribution* $\iota'_{init}(\langle s, i \rangle) = \iota_{init}(s) \cdot |I|^{-1}$*, transitions* $p'(\langle s, i \rangle, a)(\langle s', i \rangle) = p_i(s, a)(s')$*, observations* $Z = S$*, and observation function* $O(\langle s, i \rangle) = s$*.*

A policy may observe the state s but not in which MDP we are. This forces any observation-based policy to take the same choice in all environments.

Lemma 1. *Given MEMDP* $\mathcal{N}$*, there exists a winning policy iff there exists an observation-based policy* σ *such that* $\Pr_{\mathcal{N}_\sqcup}(T \mid \sigma) = 1$*.*

The statement follows as, first, any observation-based policy of the POMDP can be applied to the MEMDP, second, vice versa, any MEMDP policy is observation-based, and third, the induced MCs under these policies are isomorphic.

3.2 Belief-observation MDPs

For POMDPs, memoryless policies are not sufficient, which makes computing policies intricate. We therefore add the information that the history — i.e., the path until some point — contains. In MEMDPs, this information is the *(environment-)belief (support)* $J \subseteq I$, as the set of environments that are consistent with a path in the MEMDP. Given a belief $J \subseteq I$ and a state-action-state transition $s \xrightarrow{a} s'$, then we define $\mathsf{Up}(J, s, a, s') = \{i \in J \mid p_i(s, a, s') > 0\}$, i.e., the subset of environments in which the transition exists. For a path $\pi \in \textsc{Path}$, we define its corresponding belief $\mathcal{B}(\pi) \subseteq I$ recursively as:

$$\mathcal{B}(s_0) = I \quad \text{and} \quad \mathcal{B}(\pi \cdot sas') = \mathsf{Up}(\mathcal{B}(\pi \cdot s), s, a, s')$$

The belief in a MEMDP monotonically decreases along a path, i.e., if we know that we are not in a particular environment, this remains true indefinitely.

We aim to use a model where memoryless policies suffice. To that end, we cast MEMDPs into the exponentially larger belief-observation MDPs [16][2].

Definition 6 (BOMDP). *For a MEMDP* $\mathcal{N} = \langle S, A, \iota_{init}, \{p_i\}_{i \in I} \rangle$*, we define its* belief-observation MDP *(BOMDP) as a POMDP* $\mathcal{G}_\mathcal{N} = \langle \langle S', A, \iota'_{init}, p' \rangle, Z, O \rangle$ *with states* $S' = S \times I \times \mathcal{P}(I)$*, initial distribution* $\iota'_{init}(\langle s, j, I \rangle) = \iota_{init}(s) \cdot |I|^{-1}$*, transition relation* $p'(\langle s, j, J \rangle, a)(\langle s', j, J' \rangle) = p_j(s, a, s')$ *with* $J' = \mathsf{Up}(J, s, a, s')$*, observations* $Z = S \times \mathcal{P}(I)$*, and observation function* $O(\langle s, j, J \rangle) = \langle s, J \rangle$*.*

Compared to the union-POMDP, BOMDPs also track the belief by updating it accordingly. We clarify the correspondence between paths of the BOMDP and the MEMDP. For a path π through the MEMDP, we can mimic this path exactly in the MDPs $\mathcal{N}_j$ for $j \in \mathcal{B}(\pi)$. As we track $\mathcal{B}(\pi)$ in the state, we can deduce from the BOMDP state in which environments we can be.

Lemma 2. *For MEMDP* $\mathcal{N}$ *and the path* $\langle s_1, j, J_1 \rangle a_1 \langle s_2, j, J_2 \rangle \dots \langle s_n, j, J_n \rangle$ *of the BOMDP* $\mathcal{G}_\mathcal{N}$*, let* $j \in J_1$*. Then:* $J_n \neq \emptyset$ *and the path* $s_1 a_1 \dots s_n$ *exists in MDP* $\mathcal{N}_i$ *iff* $i \in J_1 \cap J_n$*.*

Consequently, the belief of a path can be uniquely determined by the observation of the last state reached, hence the name belief-observation MDPs.

Lemma 3. *For every pair of paths* π, π' *in a BOMDP, we have:*

$$\mathcal{B}(\pi) = \mathcal{B}(\pi') \quad \textit{implies} \quad O(\textit{last}(\pi)) = O(\textit{last}(\pi')).$$

For notation, we define $S_J = \{\langle s, j, J \rangle \mid j \in J, s \in S\}$, and analogously write $Z_J = \{\langle s, J \rangle \mid s \in S\}$. We lift the target states T to states in the BOMDP: $T_{\mathcal{G}_\mathcal{N}} = \{\langle s, j, J \rangle \mid s \in T, J \subseteq I, j \in J\}$ and define target observations $T_Z = O(T_{\mathcal{G}_\mathcal{N}})$.

Definition 7 (Winning in a BOMDP). *Let* $\mathcal{G}_\mathcal{N}$ *be a BOMDP with target observations* T_Z*. An observation-based policy* σ *is winning from some observation* $z \in Z$*, if for all* $s \in O^{-1}(z)$ *it holds that* $\Pr_{\mathcal{G}_\mathcal{N}}(s \to O^{-1}(T_Z) \mid \sigma) = 1$*.*

Furthermore, a policy σ is *winning* if it is winning for the initial distribution ι_{init}. An *observation* z *is winning* if there exists a winning policy for z. The *winning region* $\mathsf{Win}^T_{\mathcal{G}_\mathcal{N}}$ is the set of all winning observations.
Almost-sure winning in the BOMDP corresponds to winning in the MEMDP.

Theorem 1. *There exists a winning policy for a MEMDP* $\mathcal{N}$ *with target states* T *iff there exists a winning policy in the BOMDP* $\mathcal{G}_\mathcal{N}$ *with target states* $T_{\mathcal{G}_\mathcal{N}}$*.*

Intuitively, the important aspect is that for almost-sure reachability, observation-based memoryless policies are sufficient [13]. For any such policy, the induced Markov chains on the union-POMDP and the BOMDP are bisimilar [16].

BOMDPs make policy search conceptually easier. First, as memoryless policies suffice for almost-sure reachability, winning regions are independent of fixed policies: For policies σ and σ' that are winning in observation z and z', respectively, there must exist a policy $\hat{\sigma}$ that is winning for both z and z'. Second, winning regions can be determined in polynomial time in the size of the BOMDP [16].

[2] This translation is notationally simpler than going via the union-POMDP.

3.3 Fragments of BOMDPs

To avoid storing the exponentially sized BOMDP, we only build fragments: We may select any set of observations as *frontier* observations and make the states with those observations absorbing. We later discuss the selection of frontiers.

Definition 8 (Sliced BOMDP). *For a BOMDP* $\mathcal{G}_{\mathcal{N}} = \langle\langle S, A, \iota_{init}, p\rangle, Z, O\rangle$ *and a set of* frontier observations $F \subseteq Z$*, we define a BOMDP* $\mathcal{G}_{\mathcal{N}}|F = \langle\langle S, A, \iota_{init}, p'\rangle, Z, O\rangle$ *with:*

$$\forall s \in S, a \in A\colon p'(s,a) = \begin{cases} \mathsf{dirac}(s) & \text{if } O(s) \in F, \\ p(s,a) & \text{otherwise.} \end{cases}$$

We exploit this sliced BOMDP to derive constraints on the set of winning states.

Lemma 4. *For every BOMDP* $\mathcal{G}_{\mathcal{N}}$ *with states* S *and targets* T *and for all frontier observations* $F \subseteq Z$ *it holds that:* $\mathsf{Win}^{T}_{\mathcal{G}_{\mathcal{N}}|F} \subseteq \mathsf{Win}^{T}_{\mathcal{G}_{\mathcal{N}}} \subseteq \mathsf{Win}^{T \cup F}_{\mathcal{G}_{\mathcal{N}}|F}$.

Making (non-target) observations absorbing extends the set of losing observations, while adding target states extends the set of winning observations.

4 Computational Complexity

The BOMDP $\mathcal{G}_{\mathcal{N}}$ above yields an exponential time *and space* algorithm via Theorem 1. We can avoid the exponential memory requirement. This section shows the PSPACE-completeness of deciding whether a winning policy exists.

Theorem 2. *The almost-sure reachability decision problem is PSPACE-complete.*

The result follows from Lemmas 11 and 10 below. In Section 4.3, we show that representing the winning policy itself may however require exponential space.

4.1 Deciding Almost-Sure Winning for MEMDPs in PSPACE

We develop an algorithm with a polynomial memory footprint. The algorithm exploits locality of cyclic behavior in the BOMDP, as formalized by an acyclic *environment graph* and *local BOMDPs* that match the nodes in the environment graph. The algorithm recurses on the environment graph while memorizing results from polynomially many local BOMDPs.

The graph-structure of BOMDPs. First, along a path of the MEMDP, we will only gain information and are thus able to rule out certain environments [14]. Due to the monotonicity of the update operator, we have for any BOMDP that $\langle s, j, J\rangle \in \mathsf{Reachable}(\langle s', j, J'\rangle)$ implies $J \subseteq J'$. We define a graph over environment sets that describes how the belief-support can update over a run.

Definition 9 (Environment graph). *Let* $\mathcal{N}$ *be a MEMDP and* p *the transition function of* $\mathcal{G}_{\mathcal{N}}$*. The* environment graph $GE_{\mathcal{N}} = (V_{\mathcal{N}}, E_{\mathcal{N}})$ *for* $\mathcal{N}$ *is a directed graph with vertices* $V_{\mathcal{N}} = \mathcal{P}(I)$ *and edges*

$$E_{\mathcal{N}} = \{\langle J, J'\rangle \mid \exists s, s' \in S, a \in A, j \in I. p(\langle s, j, J\rangle, a, \langle s', j, J'\rangle) > 0 \text{ and } J \neq J'\}.$$

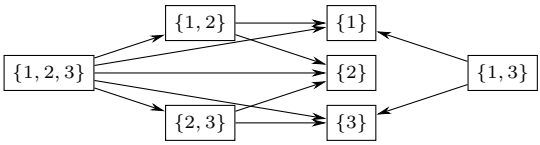


Fig. 2: The environment graph for our running example.

Example 2. Figure 2 shows the environment graph for the MEMDP in Ex. 1. It consists of the different belief-supports. For example, the transition from $\{1,2,3\}$ to $\{2,3\}$ and to $\{1\}$ is due to the action q_1 in state s_0, as shown in Fig. 1. ■

Paths in the environment graph abstract paths in the BOMDP. Path fragments where the belief-support remains unchanged are summarized into one step, as we do not create edges of the form $\langle J, J\rangle$. We formalize this idea: Let $\pi = \langle s_1, j, J_1\rangle a_1 \langle s_2, j, J_2\rangle \dots \langle s_n, j, J_n\rangle$ be a path in the BOMDP. For any $J \subseteq I$, we call π a *J-local path*, if $J_i = J$ for all $i \in [n]$.

Lemma 5. *For a MEMDP $\mathcal{N}$ with environment graph $GE_{\mathcal{N}}$, there is a path $J_1 \dots J_n$ iff there is a path $\pi = \pi_1 \dots \pi_n$ in $\mathcal{G}_{\mathcal{N}}$ s.t. every π_i is J_i-local.*

The shape of the environment graph is crucial for the algorithm we develop.

Lemma 6. *Let $GE_{\mathcal{N}} = (V_{\mathcal{N}}, E_{\mathcal{N}})$ be an environment graph for MEMDP $\mathcal{N}$. First, $E_{\mathcal{N}}(J, J')$ implies $J' \subsetneq J$. Thus, G is acyclic and has maximal path length $|I|$. The maximal outdegree of the graph is $|S|^2|A|$.*

The monotonicity regarding J, J' follows from definition of the belief update. The bound on the outdegree is a consequence from Lemma 9 below.

Local belief-support BOMDPs. Before we continue, we remark that the (future) dynamics in a BOMDP only depend on the current state and set of environments. More formally, we capture this intuition as follows.

Lemma 7. *Let $\mathcal{G}_{\mathcal{N}}$ be a BOMDP with states S'. For any state $\langle s, j, J\rangle \in S'$, let $\mathcal{N}' = \mathsf{ReachFragment}(\mathcal{N}_{\downarrow J}, \mathsf{dirac}(s))$ and $Y = \{\langle s, i, J\rangle \mid i \in J\}$. Then:*

$$\mathsf{ReachFragment}(\mathcal{G}_{\mathcal{N}}, \mathsf{unif}(Y)) = \mathcal{G}_{\mathcal{N}'}.$$

The key insight is that restricting the MEMDP does not change the transition functions for the environments $j \in J$. Furthermore, using monotonicity of the update, we only reach BOMDP-states whose behavior is determined by the environments in J.

This intuition allows us to analyze the BOMDP locally and lift the results to the complete BOMDP. We define a local BOMDP as the part of a BOMDP starting in any state in S_J. All observations not in Z_J are made absorbing.

Definition 10 (Local BOMDP). *Given a MEMDP $\mathcal{N}$ with BOMDP $\mathcal{G}_{\mathcal{N}}$ and a set of environments J. The* local BOMDP *for environments J is the fragment*

$$\textsc{Loc}\mathcal{G}(J) = \mathsf{ReachFragment}(\mathcal{G}_{\mathcal{N}_{\downarrow J}} | F, \mathsf{unif}(S_J)) \quad \textit{where} \quad F = Z \setminus Z_J \ .$$

Algorithm 1 Search algorithm

1: **function** SEARCH(MEMDP $\mathcal{N} = \langle S, A, \{p_i\}_{i \in I}, \iota_{\text{init}} \rangle$, $J \subseteq I$, $T \subseteq S$)
2: $T' \leftarrow \{\langle s, j, J \rangle \mid j \in J, s \in T\}$
3: **for** J' s.t. $E_{\mathcal{N}}(J, J')$ **do** ▷ Consider the edges in the env. graph (Def. 9)
4: $W_{J'} \leftarrow$ SEARCH$(\mathcal{N}, J', T)$ ▷ Recursion!
5: $T' \leftarrow T' \cup \{\langle s, j, J' \rangle \mid j \in J, \langle s, J' \rangle \in W_{J'}\}$
6: **return** $\mathsf{Win}^{T'}_{\text{LOC}\mathcal{G}(J)} \cap Z_J$ ▷ Construct BOMDP as in Def. 10, then model check
7:
8: **function** ASWINNING(MEMDP $\mathcal{N} = \langle S, A, \{p_i\}_{i \in I}, \iota_{\text{init}} \rangle$, $T \subseteq S$)
9: **return** $O(\mathit{Supp}(\iota_{\text{init}})) \subseteq$ SEARCH$(\mathcal{N}, I, T)$

This definition of a local BOMDP coincides with a fragment of the complete BOMDP. We then mark exactly the winning observations restricted to the environment sets $J' \subsetneq J$ as winning in the local BOMDP and compute all winning observations in the local BOMDP. These observations are winning in the complete BOMDP. The following concretization of Lemma 4 formalizes this.

Lemma 8. *Consider a MEMDP $\mathcal{N}$ and a subset of environments J.*

$$\mathsf{Win}^{T'_{\mathcal{G}_\mathcal{N}}}_{\text{LOC}\mathcal{G}(J)} \cap Z_J \;=\; \mathsf{Win}^{T_{\mathcal{G}_\mathcal{N}}}_{\mathcal{G}_\mathcal{N}} \cap Z_J \quad \textit{with} \quad T'_{\mathcal{G}_\mathcal{N}} = T_{\mathcal{G}_\mathcal{N}} \cup (\mathsf{Win}^{T_{\mathcal{G}_\mathcal{N}}}_{\mathcal{G}_\mathcal{N}} \setminus Z_J).$$

Furthermore, local BOMDPs are polynomially bounded in the size of the MEMDP.

Lemma 9. *Let $\mathcal{N}$ be a MEMDP with states S and actions A.* LOC$\mathcal{G}(J)$ *has at most $\mathcal{O}(|S|^2 \cdot |A| \cdot |J|)$ states and $\mathcal{O}(|S|^2 \cdot |A| \cdot |J|^2)$ transitions*[3]*.*

A PSPACE algorithm. We present Algorithm 1 for the MEMDP **decision problem**, which recurses depth-first over the paths in the environment graph[4]. We first state the correctness and the space complexity of this algorithm.

Lemma 10. ASWINNING *in Alg. 1 solves the decision problem in PSPACE.*

To prove correctness, we first note that SEARCH$(\mathcal{N}, J, T)$ computes $\mathsf{Win}^{T_{\mathcal{G}_\mathcal{N}}}_{\mathcal{G}_\mathcal{N}} \cap Z_J$. We show this by induction over the structure of the environment graph. For all J without outgoing edges, the local BOMDP coincides with a BOMDP just for environments J (Lemma 7). Otherwise, observe that T' in line 5 coincides with its definition in Lemma 8 and thus, by the same lemma, we return $\mathsf{Win}^{T_{\mathcal{G}_\mathcal{N}}}_{\mathcal{G}_\mathcal{N}} \cap Z_J$. To finalize the proof, a winning policy exists in the MEMDP if the observation of the initial states of the BOMDP are winning (Theorem 1). The algorithm must terminate as it recurses over all paths of a finite acyclic graph, see Lemma 6. Following Lemma 9, the number of frontier states is then bounded by $|S|^2 \cdot |A|$. The main body of the algorithm therefore requires polynomial space, and the maximal recursion depth (stack height) is $|I|$ (Lemma 6). Together, this yields a space complexity in $\mathcal{O}(|S|^2 \cdot |A| \cdot |I|^2)$.

[3] The number of transitions is the number of nonzero entries in p

[4] In contrast to depth-first-search, we do not memorize nodes we visited earlier.

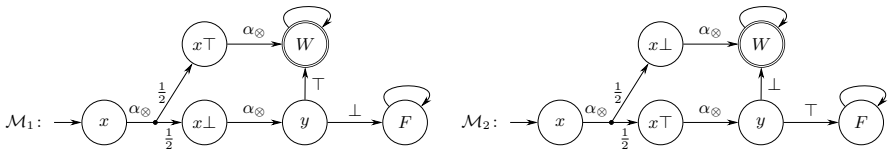


Fig. 3: Constructed MEMDP for the QBF formula $\forall x \exists y \big[(x \vee y) \wedge (\neg x \vee \neg y)\big]$.

4.2 Deciding Almost-Sure Winning for MEMDPs Is PSPACE-hard

It is not possible to improve the algorithm beyond PSPACE.

Lemma 11. *The MEMDP decision problem is PSPACE-hard.*

Hardness holds even for acyclic MEMDPs and uses the following fact.

Lemma 12. *If a winning policy exists for an acyclic MEMDP, there also exists a winning policy that is deterministic.*

In particular, almost-sure reachability coincides with avoiding the sink states. This is a safety property. For safety, deterministic policies are sufficient, as randomization visits only additional states, which is not beneficial for safety.

Regarding Lemma 11, we sketch a polynomial-time reduction from the PSPACE-complete TQBF problem [20] problem to the MEMDP decision problem. Let Ψ be a QBF formula, $\Psi = \exists x_1 \forall y_1 \exists x_2 \forall y_2 \ldots \exists x_n \forall y_n \big[\Phi\big]$ with Φ a Boolean formula in conjunctive normal form. The problem is to decide whether Ψ is true.

Example 3. Consider the QBF formula $\Psi = \forall x \exists y \big[(x \vee y) \wedge (\neg x \vee \neg y)\big]$. We construct a MEMDP with an environment for every clause, see Figure 3[5]. The state space consists of three states for each variable $v \in V$: the state v and the states $v\top$ and $v\bot$ that encode their assignment. Additionally, we have a dedicated target W and sink state F. We consider three actions: The actions *true* ($\top$) and *false* ($\bot$) semantically describe the assignment to existentially quantified variables. The action *any* $\alpha_\otimes$ is used for all other states. Every environment reaches the target state iff one literal in the clause is assigned true.

In the example, intuitively, a policy should assign the negation of x to y. Formally, the policy σ, characterized by $\sigma(\pi \cdot y) = \top$ iff $x_\bot \in \pi$, is winning. ■

As a consequence of this construction, we may also deduce the following theorem.

Theorem 3. *Deciding whether a memoryless winning policy exists is NP-complete.*

The proof of NP hardness uses a similar construction for the propositional SAT fragment of QBF, without universal quantifiers. Additionally, the problem for memoryless policies is in NP, because one can nondeterministically guess a (polynomially sized) memoryless policy and verify in each environment independently.

[5] We depict a slightly simplified MEMDP for conciseness.

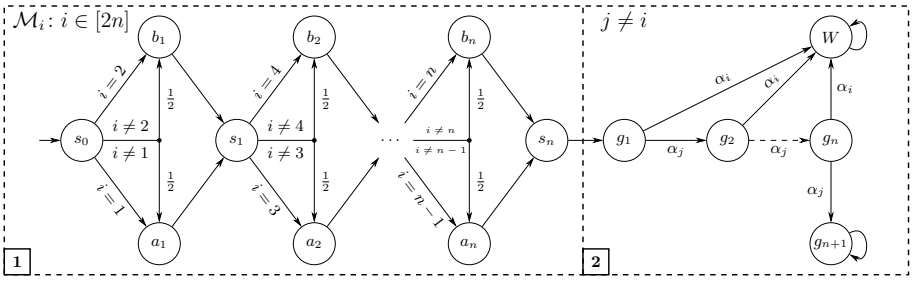


Fig. 4: Witness for exponential memory requirement for winning policies.

4.3 Policy Problem

Policies, mapping histories to actions, are generally infinite objects. However, we may extract winning policies from the BOMDP, which is (only) exponential in the MEMDP. Finite state controllers [34] are a suitable and widespread representation of policies that require only a finite amount of memory. Intuitively, the number of memory states reflects the number of equivalence classes of histories that a policy can distinguish. In general, we cannot hope to find smaller policies than those obtained via a BOMDP.

Theorem 4. *There is a family of MEMDPs* $\{\mathcal{N}^n\}_{n\geq 1}$ *where for each* n, $\mathcal{N}^n$ *has* $2n$ *environments and* $\mathcal{O}(n)$ *states and where every winning policy for* $\mathcal{N}^n$ *requires at least* 2^n *memory states.*

We illustrate the witness. Consider a family of MEMDPs $\{\mathcal{N}^n\}_n$, where $\mathcal{N}^n$ has $2n$ MDPs, $4n$ states partitioned into two parts, and at most $2n$ outgoing actions per state. We outline the MEMDP family in Figure 4. In the first part, there is only one action per state. The notation is as follows: in state s_0 and MDP $\mathcal{N}_1^n$, we transition with probability one to state a_0, whereas in $\mathcal{N}_2^n$ we transition with probability one to state b_0. In every other MDP, we transition with probability one half to either state. In state s_1, we do the analogous construction for environments 3, 4, and all others. A path $s_0 b_1 \dots$ is thus consistent with every MDP except $\mathcal{N}_1^n$. The first part ends in state s_n. By construction, there are 2^n paths ending in s_n. Each of them is (in)consistent with a unique set of n environments. In the second part, a policy may guess n times an environment by selecting an action α_i for every $i \in [2n]$. Only in MDP $\mathcal{N}_i^n$, action α_i leads to a target state. In all other MDPs, the transition leads from state g_j to g_{j+1}. The state g_{n+1} is absorbing in all MDPs. Importantly, after taking an action α_i and arriving in g_{j+1}, there is (at most) one more MDP inconsistent with the path.

Every MEMDP $\mathcal{N}^n$ in this family has a winning policy which takes $\sigma(\pi \cdot g_i) = \alpha_{2i-1}$ if $a_i \in \pi$ and $\sigma(\pi \cdot g_i) = \alpha_{2i}$ otherwise. Furthermore, when arriving in state s_n, the state of a finite memory controller must reflect the precise set of environments consistent with the history. There are 2^n such sets. The proof shows that if we store less information, two paths will lead to the same memory state, but with different sets of environments being consistent with these paths. As we

can rule out only n environments using the n actions in the second part of the MEMDP, we cannot ensure winning in every environment.

5 A Partial Game Exploration Algorithm

In this section, we present an algorithm for the policy problem. We tune the algorithm towards runtime instead of memory complexity, but aim to avoid running out of memory. We use several key ingredients to create a pragmatic variation of Alg. 1, with support for extracting the winning policy.

First, we use an abstraction from BOMDPs to a belief stochastic game (BSG) similar to [45] that reduces the number of states and simplifies the iterative construction[6]. Second, we tailor and generalize ideas from *bounded model checking* [6] to build and model check only a fragment of the BSG, using explicit *partial exploration* approaches as in, e.g., [33,9,42,29]. Third, our exploration does not continuously extend the fragment, but can also prune this fragment by using the model checking results obtained so far. The structure of the BSG as captured by the environment graph makes the approach promising and yields some natural heuristics. Fourth, the structure of the winning region allows to generalize results to unseen states. We thereby operationalize an idea from [26] in a partial exploration context. Finally, we analyze individual MDPs as an efficient and significant preprocessing step. In the following we discuss these ingredients.

Abstraction to Belief Support Games. We briefly recap stochastic games (SGs). See [38,17] for more details.

Definition 11 (SG). *A* stochastic game *is a tuple* $\mathcal{B} = \langle \mathcal{M}, S_1, S_2 \rangle$*, where* $\mathcal{M} = \langle S, A, \iota_{init}, p \rangle$ *is an MDP and* (S_1, S_2) *is a partition of* S*.*

S_1 are Player 1 states, and S_2 are Player 2 states. As common, we also 'partition' (memoryless deterministic) policies into two functions $\sigma_1 \colon S_1 \to A$ and $\sigma_1 \colon S_2 \to A$. A Player 1 policy σ_1 is winning for state s if $\Pr(T \mid \sigma_1, \sigma_2)$ for all σ_2. We (re)use $\mathsf{Win}^T_{\mathcal{B}_\mathcal{N}}$ to denote the set of states with a winning policy.

We apply a game-based abstraction to group states that have the same observation. Player 1 states capture the observation in the BOMDP, i.e., tuples $\langle s, J \rangle$ of MEMDP states s and subsets J of the environments. Player 1 selects the action a, the result is Player 2 state $\langle \langle s, J \rangle, a \rangle$. Then Player 2 chooses an environment $j \in J$, and the game mimics the outgoing transition from $\langle s, j, J \rangle$, i.e., it mimics the transition from s in $\mathcal{N}_j$. Formally:

Definition 12 (BSG). *Let* $\mathcal{G}_\mathcal{N}$ *be a BOMDP with* $\mathcal{G}_\mathcal{N} = \langle \langle S, A, \iota_{init}, p \rangle, Z, O \rangle$*. A* belief support game $\mathcal{B}_\mathcal{N}$ *for* $\mathcal{G}_\mathcal{N}$ *is an SG* $\mathcal{B}_\mathcal{N} = \langle \langle S', A', \iota'_{init}, p \rangle, S_1, S_2 \rangle$ *with* $S' = S_1 \cup S_2$ *as usual, Player 1 states* $S_1 = Z$*, Player 2 states* $S_2 = Z \times A$*, actions* $A' = A \cup I$*, initial distribution* $\iota'_{init}(\langle s, I \rangle) = \sum_{i \in I} \iota_{init}(\langle s, i, I \rangle)$*, and the (partial) transition function* p *defined separately for Player 1 and 2:*

$$p'(z, a) = \mathsf{dirac}(\langle z, a \rangle) \qquad \textit{(Player 1)}$$

$$p'(\langle z, a \rangle, j, z') = p(\langle s, j, J \rangle, a, \langle s', j, J' \rangle) \text{ with } z = \langle s, J \rangle, z' = \langle s', J' \rangle \qquad \textit{(Player 2)}$$

[6] At the time of writing, we were unaware of a polytime algorithm for BOMDPs.

Algorithm 2 Policy finding algorithm

function FindPolicy(MEMDP $\mathcal{N} = \langle S, A, \{p_i\}_{i \in I}, \iota_{\text{init}} \rangle$, targets $T \subseteq S$)
  $W \leftarrow \{\langle s, J \rangle \mid s \in T, J \subseteq I\}$; $L \leftarrow \emptyset$; $i \leftarrow 1$; $S_{\text{init}} \leftarrow \mathit{Supp}(\iota_{\text{init}}) \times \{I\}$
  while $S_{\text{init}} \cap W \neq W$ and $S_{\text{init}} \cap L = \emptyset$ **do**
    $\langle \mathcal{B}, F \rangle \leftarrow$ GenerateGameSlice($\mathcal{N}, W, L, i$)
    $W \leftarrow W \cup \mathsf{Win}_{\mathcal{B}}^{W}$
    $L \leftarrow L \cup S \setminus \mathsf{Win}_{\mathcal{B}}^{W \cup F}$
    $i \leftarrow i + 1$
  if $S_{\text{init}} \subseteq W$ **then return** ExtractPolicy(W) **else return** $\perp$

Lemma 13. *An (acyclic) MEMDP $\mathcal{N}$ with target states T is winning if(f) there exists a winning policy in the BSG $\mathcal{B}_{\mathcal{N}}$ with target states T_Z.*

Thus, on acyclic MEMDPs, a BSG-based algorithm is sound and complete, however, on cyclic MDPs, it may not find the winning policy. The remainder of the algorithm is formulated on the BSG, we use sliced BSGs as the BSG of a sliced BOMDP, or equivalently, as a BSG with some states made absorbing.

Main algorithm. We outline Algorithm 2 for the *policy problem*. We track the sets of almost-sure observations and losing observations (states in the BSG). Initially, target states are winning. Furthermore, via a simple preprocessing, we determine some winning and losing states on the individual MDPs.

We iterate until the initial state is winning or losing. Our algorithm constructs a sliced BSG and decides *on-the-fly* whether a state should be a frontier state, returning the sliced BSG and the used frontier states. We discuss the implementation below. For the sliced BSG, we compute the winning region twice: Once assuming that the frontier states are winning, once assuming they are loosing. This yields an approximation of the winning and losing states, see Lemma 4. From the winning states, we can extract a randomized winning policy [13].

Soundness. Assuming that the $\mathcal{B}_{\mathcal{N}}$ is indeed a sliced BSG with frontier F. Then the following invariant holds: $W \subseteq \mathsf{Win}_{\mathcal{B}_{\mathcal{N}}}^{T}$ and $L \cap \mathsf{Win}_{\mathcal{B}_{\mathcal{N}}}^{T} = \emptyset$. This invariant exploits that from a sliced BSG we can (implicitly) slice the complete BSG while preserving the winning status of every state, formalized below. In future iterations we only explore the implicitly sliced BSG.

Lemma 14. *Given $W \subseteq \mathsf{Win}_{\mathcal{B}_{\mathcal{N}}}^{T_{\mathcal{B}_{\mathcal{N}}}}$ and $L \subseteq S \setminus \mathsf{Win}_{\mathcal{B}_{\mathcal{N}}}^{T_{\mathcal{B}_{\mathcal{N}}}}$: $\mathsf{Win}_{\mathcal{B}_{\mathcal{N}}}^{T_{\mathcal{B}_{\mathcal{N}}}} = \mathsf{Win}_{\mathcal{B}_{\mathcal{N}}|W \cup L}^{T_{\mathcal{B}_{\mathcal{N}}} \cup W}$*

Termination depends on the sliced game generation. It suffices to ensure that in the long run, either W or L grow as there are only finitely many states. If W and L remain the same longer than some number of iterations, $W \cup L$ will be used as frontier. Then, the new game will suffice to determine if $s \in W$ in one shot.

Generating the sliced BSG. Algorithm 3 outlines the generation of the sliced BSG. In particular, we explore the implicit BSG from the initial state but make every state that we do not explicitly explore absorbing. In every iteration, we first check if there are states in Q left to explore and if the number of explored states

Algorithm 3 Game generation algorithm

```
1: function GenerateGameSlice(MEMDP 𝒩, W, L, i)
2:     Q ← {s_ι}; E = {s_ι}
3:     while s ∈ Q and |E| ≤ Bound[i] exists do
4:         E ← E ∪ {s}                                   ▷ Mark s as explored
5:         ℬ ← 𝒩_𝒩|(S \ E)          ▷ Extend game, cut-off everything not explored
6:         Q ← Reachable(ℬ) \ (E ∪ W ∪ L)                ▷ Add newly reached states
7:     return ℬ, Q
```

in E is below a threshold $\mathsf{Bound}[i]$. Then, we take a state from the priority queue and add it to E. We find new reachable states[7] and add them to the queue Q.

Generalizing the winning and losing states. We aim to determine that a state in the game $\mathcal{B}_\mathcal{N}$ is winning without ever exploring it. First, observe:

Lemma 15. *A winning policy in MEMDP $\mathcal{N}$ is winning in $\mathcal{N}_{\downarrow J}$ for any J.*

A direct consequence is the following statement for two environments $J_1 \subseteq J_2$:

$$\langle s, J_2\rangle \in \mathsf{Win}^T_{\mathcal{B}_\mathcal{N}} \quad \text{implies} \quad \langle s, J_1\rangle \in \mathsf{Win}^T_{\mathcal{B}_\mathcal{N}}.$$

Consequently, we can store W (and symmetrically, L) as follows. For every MEMDP state $s \in S$, $W_s = \{J \mid \langle s, J\rangle \in W\}$ is downward closed on the partial order $P = (I, \subset)$. This allows for efficient storage: We only have to store the set of pairwise maximal elements, i.e., the antichain,

$$W_s^{\max} = \{J \in W_s \mid \forall J' \in W_s \text{ with } J \not\subseteq J'\}.$$

To determine whether $\langle s, J\rangle$ is winning, we check whether $J \subseteq J'$ for some $J' \in W_s^{\max}$. Adding J to $W_s^{\max}$ requires removing all $J' \subseteq J$ and then adding J. Note, however, that $|W_s^{\max}|$ is still exponential in $|I|$ in the worst case.

Selection of heuristics. The algorithm allows some degrees of freedom. We evaluate the following aspects empirically. (1) The maximal size $\texttt{bound}[i]$ of a sliced BSG at iteration i is critical. If it is too small, the sets W and L will grow slowly in every iteration. The trade-off is further complicated by the fact that the sets W and L may generalize to unseen states. (2) For a fixed $\texttt{bound}[i]$, it is unclear how to prioritize the exploration of states. The PSPACE algorithm suggests that going deep is good, whereas the potential for generalization to unseen states is largest when going broad. (3) Finally, there is overhead in computing both W and L. If there is a winning policy, we only need to compute W. However, computing L may ensure that we can prune parts of the state space. A similar observation holds for computing W on unsatisfiable instances.

Remark 1. Algorithm 2 can be mildly tweaked to meet the PSPACE algorithm in Algorithm 1. The priority queue must ensure to always include complete

[7] In l. 5 we do not rebuild the game $\mathcal{B}$ from scratch but incrementally construct the data structures. Likewise, reachable states are a direct byproduct of this construction.

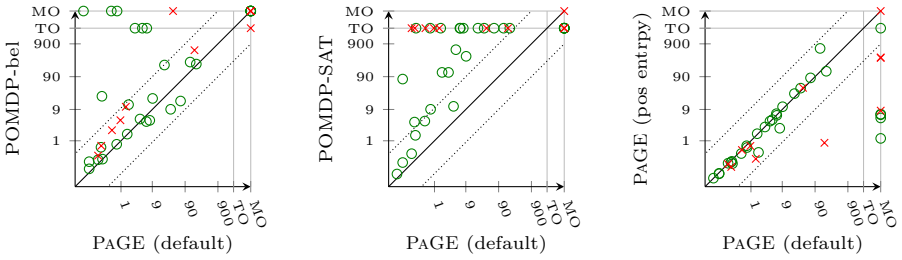


Fig. 5: Performance of baselines and novel PAGE algorithm

(reachable) local BSGs and to explore states $\langle s, J \rangle$ with small J first. Furthermore, W and L require regular pruning, and we cannot extract a policy if we prune W to a polynomial size bound. Practically, we may write pruned parts of W to disk.

6 Experiments

We highlight two aspects: (1) A comparison of our prototype to existing baselines for POMDPs, and (2) an examination of the exploration heuristics. The technical report [41] contains details on the implementation, the benchmarks, and more results.

Implementation. We provide a novel *PArtial Game Exploration* (PAGE) prototype, based on Algorithm 2, on top of the probabilistic model checker STORM [22]. We represent MEMDPs using the PRISM language with integer constants. Every assignment to these constants induces an explicit MDP. SGs are constructed and solved using existing data structures and graph algorithms.

Setup. We create a set of benchmarks inspired by the POMDP and MEMDP literature [26,12,21]. We consider a combination of satisfiable and unsatisfiable benchmarks. In the latter case, a winning policy does not exist. We construct POMDPs from MEMDPs as in Definition 5. As baselines, we use the following two existing POMDP algorithms. For almost-sure properties, a *belief-MDP construction* [7] acts similar to an efficiently engineered variant of our game-construction, but tailored towards more general quantitative properties. A *SAT-based approach* [26] aims to find increasingly larger policies. We evaluate all benchmarks on a system with a 3GHz Intel Core i9-10980XE processor. We use a time limit of 30 minutes and a memory limit of 32 GB.

Results. Figure 5 shows the (log scale) performance comparisons between different configurations[8]. Green circles reflect satisfiable and red crosses unsatisfiable benchmarks. On the x-axis is PAGE in its default configuration. The first plot compares to the belief-MDP construction. The tailored heuristics and representation of the belief-support give a significant edge in almost all cases. The few points

[8] Every point $\langle x, y \rangle$ in the graph reflects a benchmarks which was solved by the configuration on the x-axis in x time and by the configuration on the y-axis in y time. Points above the diagonal are thus faster for the configuration on the x-axis.

Table 1: Satisfiable and unsatisfiable benchmark results

				PAGE(posentr)		PAGE(negentr)		Belief	SAT
	$\|I\|$	$\|S\|$	$\|A\|$	t	n	t	n	t	t
Grid	19	132	4	0.2	3002	0.2	3002	0.6	3.7
	39	152	4	0.4	9007	1.6	41029	12.6	121.3
	199	474	4	6.4	337177	MO		MO	TO
Catch	256	625	4	6.6	93614	5.9	41094	3.8	TO
	256	6561	4	40.1	749295	32.6	337899	9.1	TO
	256	14641	4	82.5	1826922	65.3	338079	16.2	TO
Exp	8	19	9	0.1	349	0.1	349	0.1	75.9
	20	43	21	131.4	192163	197.6	448443	217.6	TO
	24	51	25	TO		MO		MO	TO
Frogger	10	1200	4	0.2	1200	0.2	1200	22.7	1.4
	20	1200	4	0.4	1200	0.5	1200	MO	3.9
	80	4000	4	4.4	4000	4.4	4000	TO	597.3
	99	4000	4	5.9	8001	6.1	8001	TO	TO

				PAGE(posentr)		PAGE(negentr)		Belief
	$\|I\|$	$\|S\|$	$\|A\|$	t	n	t	n	t
MMind	16	21	16	0.1	1003	0.2	1445	0.3
	27	17	27	0.5	5167	0.5	7579	2.0
	32	25	32	0.6	7799	0.9	11809	4.2
	81	21	81	41.1	170291	38.6	296407	MO
Exp	20	42	21	0.8	9005	173.8	388127	576.1
	24	50	25	8.3	41022	MO		MO
	32	66	33	347.7	337177	MO		MO

below the line are due to a higher exploration rate when building the state space. The second plot compares to the SAT-based approach, which is only suitable for finding policies, not for disproving their existence. This approach implicitly searches for a particular class of policies, whose structure is not appropriate for some MEMDPs. The third plot compares PAGE in the default configuration – with negative entropy as priority function – with PAGE using positive entropy. As expected, different priorities have a significant impact on the performance.

Table 1 shows an overview of satisfiable and unsatisfiable benchmarks. Each table shows the number of environments, states, and actions-per-state in the MEMDP. For PAGE, we include both the default configuration (negative entropy) and variation (positive entropy). For both configurations, we provide columns with the time and the maximum size of the BSG constructed. We also include the time for the two baselines. Unsurprisingly, the number of states to be explored is a good predictor for the performance and the relative performance is as in Fig. 5.

7 Conclusion

This paper considers multi-environment MDPs with an arbitrary number of environments and an almost-sure reachability objective. We show novel and tight complexity bounds and use these insights to derive a new algorithm. This algorithm outperforms approaches for POMDPs on a broad set of benchmarks. For future work, we will apply an algorithm directly on the BOMDP [16].

Data-Availability Statement

Supplementary material related to this paper is openly available on Zenodo at: https://doi.org/10.5281/zenodo.7560675

References

1. Roman Andriushchenko, Milan Ceska, Sebastian Junges, Joost-Pieter Katoen, and Simon Stupinský. PAYNT: A tool for inductive synthesis of probabilistic programs. In *CAV*, volume 12759 of *LNCS*, pages 856–869. Springer, 2021.
2. Sebastian Arming, Ezio Bartocci, Krishnendu Chatterjee, Joost-Pieter Katoen, and Ana Sokolova. Parameter-independent strategies for pmdps via pomdps. In *QEST*, volume 11024 of *LNCS*, pages 53–70. Springer, 2018.
3. Mohammad Gheshlaghi Azar, Alessandro Lazaric, and Emma Brunskill. Sequential transfer in multi-armed bandit with finite set of models. In *NIPS*, pages 2220–2228, 2013.
4. Christel Baier, Marcus Größer, and Nathalie Bertrand. Probabilistic ω-automata. *J. ACM*, 59(1):1:1–1:52, 2012.
5. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
6. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003.
7. Alexander Bork, Sebastian Junges, Joost-Pieter Katoen, and Tim Quatmann. Verification of indefinite-horizon pomdps. In *ATVA*, volume 12302 of *LNCS*, pages 288–304. Springer, 2020.
8. Alexander Bork, Joost-Pieter Katoen, and Tim Quatmann. Under-approximating expected total rewards in pomdps. In *TACAS (2)*, volume 13244 of *LNCS*, pages 22–40. Springer, 2022.
9. Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Kretínský, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. Verification of markov decision processes using learning algorithms. In *ATVA*, volume 8837 of *LNCS*, pages 98–114. Springer, 2014.
10. Peter Buchholz and Dimitri Scheftelowitsch. Computation of weighted sums of rewards for concurrent mdps. *Math. Methods Oper. Res.*, 89(1):1–42, 2019.
11. Iadine Chades, Josie Carwardine, Tara G. Martin, Samuel Nicol, Régis Sabbadin, and Olivier Buffet. Momdps: A solution for modelling adaptive management problems. In *AAAI*. AAAI Press, 2012.
12. Krishnendu Chatterjee, Martin Chmelik, and Jessica Davies. A symbolic sat-based algorithm for almost-sure reachability with small strategies in pomdps. In *AAAI*, pages 3225–3232. AAAI Press, 2016.
13. Krishnendu Chatterjee, Martin Chmelik, Raghav Gupta, and Ayush Kanodia. Optimal cost almost-sure reachability in pomdps. *Artif. Intell.*, 234:26–48, 2016.
14. Krishnendu Chatterjee, Martin Chmelík, Deep Karkhanis, Petr Novotný, and Amélie Royer. Multiple-environment markov decision processes: Efficient analysis and applications. In *ICAPS*, pages 48–56. AAAI Press, 2020.
15. Krishnendu Chatterjee, Martin Chmelik, and Mathieu Tracol. What is decidable about partially observable markov decision processes with omega-regular objectives. In *CSL*, volume 23 of *LIPIcs*, pages 165–180. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

16. Krishnendu Chatterjee, Martin Chmelik, and Mathieu Tracol. What is decidable about partially observable markov decision processes with ω-regular objectives. *J. Comput. Syst. Sci.*, 82(5):878–911, 2016.
17. Krishnendu Chatterjee, Marcin Jurdzinski, and Thomas A. Henzinger. Simple stochastic parity games. In *CSL*, volume 2803 of *LNCS*, pages 100–113. Springer, 2003.
18. Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. Profeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects Comput.*, 30(1):45–75, 2018.
19. Luca de Alfaro. The verification of probabilistic systems under memoryless partial-information policies is hard. Technical report, UC Berkeley, 1999. Presented at ProbMiV.
20. M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.
21. Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. The quantitative verification benchmark set. In *TACAS (1)*, volume 11427 of *LNCS*, pages 344–350. Springer, 2019.
22. Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *Int. J. Softw. Tools Technol. Transf.*, 24(4):589–610, 2022.
23. Manfred Jaeger, Giorgio Bacci, Giovanni Bacci, Kim Guldstrand Larsen, and Peter Gjøl Jensen. Approximating Euclidean by Imprecise Markov Decision Processes. In *ISoLA (1)*, volume 12476 of *LNCS*, pages 275–289. Springer, 2020.
24. Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Parameter synthesis in markov models: A gentle survey. *CoRR*, abs/2207.06801, 2022.
25. Bengt Jonsson and Kim Guldstrand Larsen. Specification and refinement of probabilistic processes. In *LICS*, pages 266–277. IEEE Computer Society, 1991.
26. Sebastian Junges, Nils Jansen, and Sanjit A. Seshia. Enforcing almost-sure reachability in pomdps. In *CAV (2)*, volume 12760 of *LNCS*, pages 602–625. Springer, 2021.
27. Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.
28. Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning. *CoRR*, abs/2111.09794, 2021.
29. Jan Kretínský and Tobias Meggendorfer. Of cores: A partial-exploration framework for markov decision processes. *Log. Methods Comput. Sci.*, 16(4), 2020.
30. Marta Kwiatkowska, Gethin Norman, and Dave Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
31. Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In *ICML*, pages 362–370. Morgan Kaufmann, 1995.
32. Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artif. Intell.*, 147(1-2):5–34, 2003.
33. H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *ICML*, volume 119 of *ACM International Conference Proceeding Series*, pages 569–576. ACM, 2005.

34. Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Pack Kaelbling. Learning finite-state controllers for partially observable environments. In *UAI*, pages 427–436. Morgan Kaufmann, 1999.
35. Gethin Norman, David Parker, and Xueyi Zou. Verification and control of partially observable probabilistic systems. *Real Time Syst.*, 53(3):354–402, 2017.
36. Jean-François Raskin and Ocan Sankur. Multiple-environment markov decision processes. In *FSTTCS*, volume 29 of *LIPIcs*, pages 531–543. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014.
37. John H. Reif. The complexity of two-player games of incomplete information. *J. Comput. Syst. Sci.*, 29(2):274–301, 1984.
38. L. S. Shapley. Stochastic games*. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953.
39. Trey Smith and Reid G. Simmons. Point-based POMDP algorithms: Improved analysis and implementation. In *UAI*, pages 542–547. AUAI Press, 2005.
40. Lauren N. Steimle, David L. Kaufman, and Brian T. Denton. Multi-model markov decision processes. *IISE Trans.*, 53(10):1124–1139, 2021.
41. Marck van der Vegt, Nils Jansen, and Sebastian Junges. Robust almost-sure reachability in multi-environment mdps. *CoRR*, abs/2301.11296, 2023.
42. Matthias Volk, Sebastian Junges, and Joost-Pieter Katoen. Fast dynamic fault tree analysis by model checking techniques. *IEEE Trans. Ind. Informatics*, 14(1):370–379, 2018.
43. Wolfram Wiesemann, Daniel Kuhn, and Berç Rustem. Robust markov decision processes. *Math. Oper. Res.*, 38(1):153–183, 2013.
44. Tobias Winkler, Sebastian Junges, Guillermo A. Pérez, and Joost-Pieter Katoen. On the complexity of reachability in parametric markov decision processes. In *CONCUR*, volume 140 of *LIPIcs*, pages 14:1–14:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
45. Leonore Winterer, Sebastian Junges, Ralf Wimmer, Nils Jansen, Ufuk Topcu, Joost-Pieter Katoen, and Bernd Becker. Strategy synthesis for pomdps in robot planning via game-based abstractions. *IEEE Trans. Autom. Control.*, 66(3):1040–1054, 2021.

Mungojerrie: Linear-Time Objectives in Model-Free Reinforcement Learning*

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

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

Abstract. Mungojerrie is an extensible tool that provides a framework to translate linear-time objectives into reward for reinforcement learning (RL). The tool provides convergent RL algorithms for stochastic games, reference implementations of existing reward translations for ω-regular objectives, and an internal probabilistic model checker for ω-regular objectives. This functionality is modular and operates on shared data structures, which enables fast development of new translation techniques. Mungojerrie supports finite models specified in PRISM and ω-automata specified in the HOA format, with an integrated command line interface to external linear temporal logic translators. Mungojerrie is distributed with a set of benchmarks for ω-regular objectives in RL.

1 Introduction

Reinforcement learning (RL) [41] is a sequential optimization approach where a decision maker learns to optimally resolve a sequence of choices based on feedback received from the environment. This feedback often takes the form of rewards and punishments proportional to the fitness of the decisions taken by the agent (or their effects) as judged by the environment towards some higher-level objectives. We call such objectives *learning objectives*. RL is inspired by the way dopamine-driven organisms latch on to past rewarding actions and hence, historically, RL adopted a myopic way of looking at the reward sequences in the form of the discounted-sum of rewards, where the discount factor controls the weight placed toward future rewards. More recently, other forms of reward aggregation, such as limit-average, have also been considered. A key design challenge for users of RL is that of translation: given a class of learning objectives and aggregator functions, design a reward function from the sequence of learner's choices to scalar rewards such that an RL agent maximizing the aggregated sum of rewards converges to an optimal policy for the learning objective.

* Mungojerrie is available at plv.colorado.edu/mungojerrie. This work is supported in part by the National Science Foundation (NSF) grant CCF-2009022 and by NSF CAREER award CCF-2146563. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No 864075 (CAESAR) and 956123 (FOCETA).

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 527–545, 2023.
https://doi.org/10.1007/978-3-031-30823-9_27

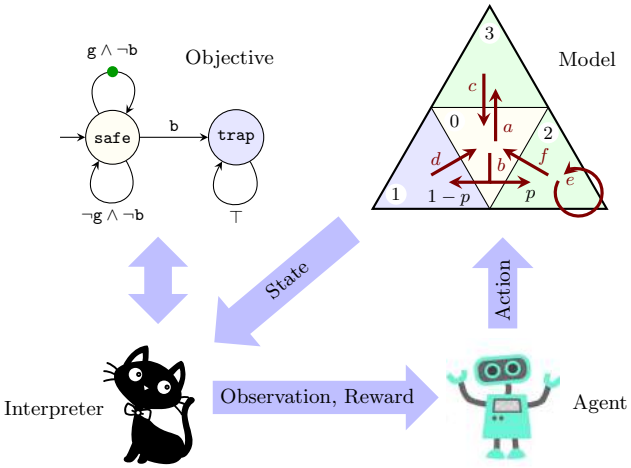


Fig. 1. The reinforcement learning loop implemented within Mungojerrie. The interpreter assigns reward to the agent based on the state of the model and automaton.

The translation of objectives to reward signals has historically been a largely manual process. Such translations not only depend on the expertise of the translator in reward engineering, they also pose obstacles to providing formal guarantees on the faithfulness of the translation. Unsurprisingly, specifying reward manually is prone to error [22,44]. As the practice of model-free RL continues to produce impressive results [38,31,29], the integration of RL in safety-critical system design is inevitable. An alternative to manually programming the reward function is to specify the objective in a formal language and have it "compiled" to a reward function. We call such a translation a *reward scheme*.

In designing reward schemes for RL, one strives to achieve an overall translation that is *faithful* (maximizing reward means maximizing the probability of achieving the objective) and *effective* (RL quickly converges to optimal strategies). While the faithfulness of a reward scheme can be established theoretically, its effectiveness requires experimental evaluation. Experimenting with reward schemes requires a framework for specifying learning objectives, environments, a wide range of RL algorithms, and an interface for connecting reward schemes with these components. In addition, it may be beneficial to have access to a probabilistic model checker to evaluate the quality of the policy computed by RL, and to compare it against ground truth.

> *Mungojerrie is designed to provide this functionality for learning requirements expressible as linear-time objectives (ω-regular languages [32] and linear temporal logic [27,33]) against finite MDPs and stochastic games.*

Features. Mungojerrie is designed with ease of use and extensibility in mind. Models in Mungojerrie can be specified in PRISM [25], which maintains compati-

bility with existing benchmarks, or by explicitly constructing the model via calls to internal functions. Mungojerrie supports reading ω-automata in the Hanoi Omega Automata (HOA) format [2], and has a command line interface connecting Mungojerrie with performant LTL translators (Spot [7] and Owl [24]). Mungojerrie provides an OpenAI Gym [4] like interface between the RL algorithms (included with the tool) and the learning environment to allow integration with off-the-shelf RL algorithms. The tool also has methods for performing probabilistic model checking (including end-component decomposition, stochastic shortest-path, and discounted-reward optimization) of ω-regular objectives on the same data structures used for learning. Mungojerrie also provides reference implementations of several reward schemes [11,12,14,19,23] proposed by the formal methods community. Mungojerrie is packaged with over 100 benchmarks and outputs GraphViz [8] for easy visualization of small models and automata.

An introductory example. Figure 2 shows an example MDP in which a gambler places bets with the aim of accumulating a wealth of 7 units. In addition the gambler will quit if her wealth wanes to just one unit more than once. This objective is captured by the (deterministic) Büchi automaton of Fig. 3. Mungojerrie computes a strategy for the gambler that maximizes the probability of satisfying her objective. Figure 4 shows the Markov chain that results from following this strategy. This figure was minimally modified from GraphViz output from Mungojerrie. Note that the strategy altogether avoids the state in which $x = 1$; hence it achieves the same probability of success $(5/7)$ as an optimal strategy for the simpler objective of eventually reaching $x = 7$ (without going broke). Mungojerrie computes the strategy of Fig. 4 by RL; it can also verify it by probabilistic model checking.

2 Overview of Mungojerrie

Models. The systems used in Mungojerrie consist of finite sets of states and actions, where states are labeled with atomic propositions. There are at most two strategic players: Max player and Min player. Each state is controlled by one player. We call models where all states are controlled by Max player Markov decision processes (MDPs) [34]. Else, we refer to them as stochastic games [5].

Mungojerrie supports parsing models specified in the PRISM language. The allowed model types are "mdp" (Markov decision process) and "smg" (stochastic multiplayer game) with two players. There should be one initial state. The interface for building the model is exposed, allowing extensions of Mungojerrie to connect with parsers for other languages. The authors of [6] used Mungojerrie in their experiments by extending the tool to support continuous-time MDPs.

Properties. The properties natively supported by Mungojerrie are ω-regular languages. Starting from the initial state, the players produce an infinite sequence of states with a corresponding infinite sequence of atomic propositions: an ω-word. The inclusion of this ω-word in our ω-regular language determines

whether or not this particular run satisfies the property. The Max player maximizes the probability that a run is satisfying, while goal of the Min player is the opposite.

We specify our ω-regular language as an ω-automaton, which may be nondeterministic. For model checking and RL, this nondeterminism must be resolved on the fly. Automata where this can be done in any MDP without changing acceptance are said to be Good-for-MDPs (GFM) [13]. Automata where this can be done in any stochastic game without changing acceptance are said to be Good-for-Games (GFG) [21]. In general, nondeterministic Büchi automata are not GFM, but two classes of GFM Büchi automata with limited nondeterminism have been studied: suitable limit-deterministic Büchi automata [10,37] and slim Büchi automata [13].

The user of Mungojerrie can either provide the ω-automaton directly or use one of the supported external translators to generate the automaton from LTL with a single call to Mungojerrie. Mungojerrie reads automata specified in the HOA format. Mungojerrie supports providing the ω-automaton directly for testing the effectiveness of different automata for learning (see Section 4). The LTL translators that can be called from Mungojerrie are the EPMC plugin from [13], Spot [7], and Owl [24] for generating slim Büchi, deterministic parity, and suitable limit-deterministic Büchi automata. The user is responsible for the ω-automata provided directly having the appropriate property, GFM or GFG.

For use in Mungojerrie, the labels and acceptance conditions for the automaton should be on the transitions. The acceptance conditions supported by

```
0  mdp
1
2  const int Wealth = 5;    // initial gambler's wealth
3  const double p   = 1/2; // probability of winning one bet
4
5  label "rich" = x = 7;
6  label "poor" = x = 1;
7
8  module gambler
9    x : [0..7] init Wealth;
10
11   [b0] x=0 ∨ x=7 → true; // absorbing states
12   [b1] x>0 ∧ x<7 → p : (x'=x+1) + (1−p) : (x'=x−1);
13   [b2] x>1 ∧ x<6 → p : (x'=x+2) + (1−p) : (x'=x−2);
14   [b3] x>2 ∧ x<5 → p : (x'=x+3) + (1−p) : (x'=x−3);
15 endmodule
```

Fig. 2. A Gambler's Ruin model in the PRISM language. Line 13, for example, says that when $1 < x < 6$, the gambler may bet two units because action `b2` is enabled. The '+' sign does double duty: as addition symbol in arithmetic expressions and as separator of probabilistic transitions.

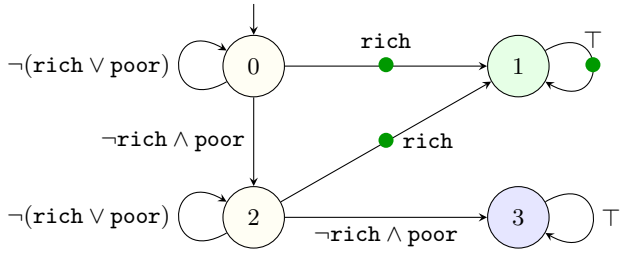


Fig. 3. Deterministic Büchi automaton equivalent to the LTL formula $\neg\texttt{poor}\,\mathsf{U}\big(\texttt{rich}\vee(\texttt{poor}\wedge\mathsf{X}(\neg\texttt{poor}\,\mathsf{U}\,\texttt{rich}))\big)$. The transitions marked with the green dots are accepting.

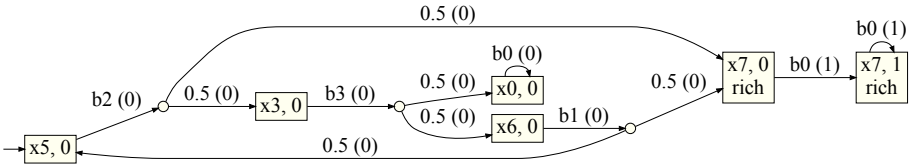


Fig. 4. Optimal gambler strategy for the objective of Fig. 3. Boxes are decision states and circles are probabilistic choice states. For a decision state, the label gives the value of x and the state of the automaton. Transitions are labelled with either an action or a probability, and with the priority (1 for accepting and 0 for non-accepting).

Mungojerrie should be reducible to parity acceptance conditions without altering the transition structure of the automaton. This includes parity, Büchi, co-Büchi, Streett 1 (one pair), and Rabin 1 (one pair) conditions. Nondeterministic automata must have Büchi acceptance conditions. Generalized acceptance conditions are not supported in version 1.1.

Reinforcement Learning. The RL algorithms optimize over MDP/Stochastic game environments equipped with a Markovian reward function. The reward function assigns a reward $R_{t+1} \in \mathbb{R}$ dependent on the state and action at timestep t and the next state at timestep $t+1$. As the players make their choices within the environment, the resulting play produces a sequence of states, actions, and rewards $(S_0, A_0, R_1, S_1, A_1, R_2, \ldots)$. The discounted reward aggregator is

$$\mathrm{disc}_\gamma(\pi,\nu) = \mathbb{E}_{\pi,\nu}\left[\sum_{t\geq 0}\gamma^t R_{t+1}\right] ,$$

where π is the strategy for Max player, ν is the strategy for Min player, $\gamma \in [0,1)$ is the discount factor, and R_t is the reward at timestep t. We can set $\gamma = 1$ when

with probability 1 we enter an absorbing sink (termination), where we receive no reward. This is called the episodic setting. Another well-studied RL aggregator is the limit-average reward defined as

$$\mathrm{avg}(\pi, \nu) = \limsup_{n \to \infty} \frac{1}{n} \mathbb{E}_{\pi,\nu} \Big[\sum_{n \geq t \geq 0} R_{t+1} \Big].$$

The limit-average reward aggregator is natural in the continuing setting, where the agent's trajectory is never reset and there is no preferred initial state [30]. The objective of RL is to compute the optimal value and policies for a given aggregator. Mungojerrie includes the stochastic game extensions of Q-learning [43], Double Q-learning [20], and Sarsa(λ) [40] for RL in finite state and action models. Mungojerrie also includes Differential Q-learning [42] for average RL in finite communicating MDPs. We collectively refer to parameters that are set by hand prior to running an RL algorithm as hyperparameters. Mungojerrie supports changing all hyperparameters from the command line. As the design of Mungojerrie separates the learning agent(s) from the reward scheme, extending Mungojerrie to include another RL algorithm is easy.

Reward Schemes. The user of Mungojerrie can either select one of the reward schemes included with the tool or extend the tool to include a new reward scheme. Mungojerrie also allows the use of the reward specified in the PRISM model (either state- or action-based). The following reward schemes are included in version 1.1 of Mungojerrie:

– Limit-reachability. The *limit-reachability* scheme [11] uses a GFM Büchi automaton. This reward scheme converts accepting edges in the automaton into a transition to a sink with probability $1-\zeta$ with a reward of $+1$, where $0 < \zeta < 1$ is a hyperparameter. All other transitions produce zero reward. For a sufficiently large ζ and discount factor γ, strategies that are optimal for the discounted reward maximize the probability of satisfaction of the Büchi objective.
– Multi-discounted. The multi-discounted reward scheme [3] also uses a GFM Büchi automaton. This translation converts accepting edges in the automaton into a transition that gives $1-\gamma_B$ reward with a discount of γ_B, where $0 < \gamma_B < 1$ is a hyperparameter. All other transitions yield no reward and are discounted by the standard discount factor γ. For suitably large γ_B and γ, discounted reward optimal strategies maximize the probability of satisfaction of the Büchi objective.
– Dense limit-reachability. The dense limit-reachability reward scheme [12] connects the approaches of [11] and [3]. This reward scheme is identical to [11] except for giving a $+1$ reward given every time an accepting transition is seen, instead of only when the transition to the sink succeeds. Since discounting can be thought of as a constant stopping probability [41], this reward scheme is the same in expectation as a scaled version of [3].
– Parity. The parity reward scheme was proposed for stochastic games in [14]. For two-player games, it requires a GFG automaton. This translation utilizes a deterministic parity automaton with a max odd objective. Transitions of priority

i go to a sink with probability ε^{k-i}, where k is the number of priorities and $0 < \varepsilon < 1$ is a hyperparameter. The transition to the sink receives a $+1$ or -1 reward for odd or even priorities, respectively. All other transitions receive a zero reward. For sufficiently small ε, maximizing the cumulative reward results in a strategy maximizing the probability of satisfaction of the parity objective.

– Priority tracker. The priority tracker reward scheme was proposed by Hahn et al. [14]. For MDPs, Hahn et al. introduce a priority tracker gadget that takes a parity objective with a hyperparameter $0 < \varepsilon < 1$. The priority tracker consists of two stages. In stage one, we wait for transients to end by ending the stage with probability ε on each step. In the second stage, we detect the maximum priority occuring infinitely often with a set of wait states, where we accept the current maximum with probability ε on each step. For sufficiently small ε and large discount γ, maximizing the discounted reward also maximizes the probability of satisfaction of the parity objective.

– Lexicographic. Hahn et al.[19] proposed this reward scheme for lexicographic ω-regular objectives. In this reward scheme, there is a tracker gadget that keeps track of which accepting edges for the GFM Büchi automata have been seen. When the tracker indicates that at least one accepting edge has been seen, the learning agent can decide to "cash in" the tracker, which clears the tracker. When this happens, with probability $1 - \zeta$ the learning agent receives a reward which is the weighted sum of seen accepting edges, scaled by powers of f, and transitions to a terminating sink, where $0 < \zeta < 1$ and $f \geq 1$ are hyperparameters. For suitable f, ζ, and γ, maximizing the discounted reward yields the lexicographically optimal strategy.

– Average. The average reward scheme [23] translates absolute liveness ω-regular objectives, which means the objective is concerned with eventual satifaction, to average reward for communicating MDPs. Given a GFM Büchi automaton, transitions from every state in the automaton back to the initial state are introduced, so called "resets". A hyperparameter $c < 0$ is introduced which gives a penalizing reward to these resets. Accepting edges are then given a reward of $+1$. Positional policies that maximize the average reward also maximize the probability of satisfaction of the objective.

– Reward on accept. This reward scheme was proposed in [35]. The translation of [35] picks a pair in a Rabin automaton to satisfy, and gives positive and negative reward for the good and bad states of the pair, respectively. In general, picking the winning pair ahead of time is not possible [11]. For a Büchi automaton, this corresponds to giving positive $(+1)$ rewards for accepting edges and zero rewards otherwise. While this reward scheme was shown to be not faithful [11] for general objectives, it is included for comparison purposes.

3 Tool Design

The primary design goal of Mungojerrie is to enable extensibility. To accomplish this, Mungojerrie separates different processing stages as much as possible so that extensions can reuse other components. We begin by presenting the architecture

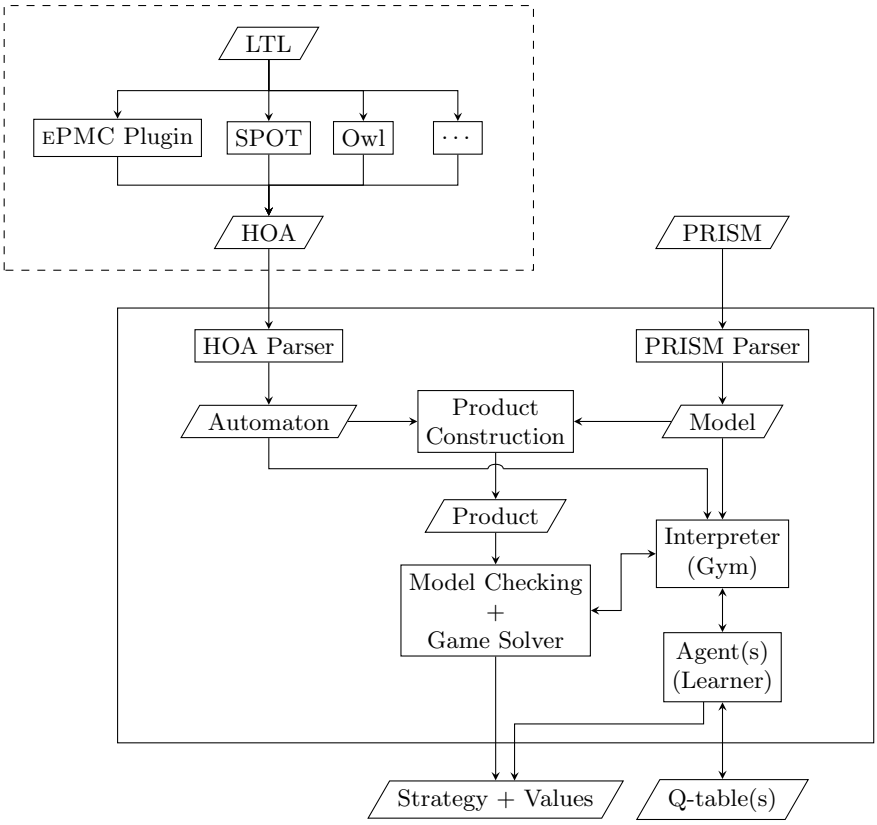


Fig. 5. Architecture of Mungojerrie 1.1.

of Mungojerrie. Afterwards, we take a closer at the novel slim Büchi automata plugin, which is described here in detail for the first time.

Architecture of Mungojerrie. Mungojerrie begins its execution by parsing the input PRISM and HOA (see upper part of Fig. 5). The HOA is either read in from a file or piped from a call to one of the supported LTL translators. In particular the ePMC plugin from [13], an LTL translator capable of producing slim Büchi automata, is packaged with the tool. Requested automaton modifications, such as determinization, are run after this step. If specified, Mungojerrie creates the synchronous product between the automaton and the model, and runs model checking or game solving [1,15,16]. The requested strategy and values are returned. Due to this step, Mungojerrie has been connected to external linear program solvers. This enabled the extension of Mungojerrie to compute reward maximizing policies via a linear program for branching Markov decision processes in [18].

If learning has been specified, the interpreter takes the automaton and model, without explicitly forming the product, and provides an interface akin to OpenAI Gym [4] for the RL agent to interact with the environment and receive rewards. When learning is complete, the Q-table(s) can be saved to a file for later use, and the interpreter forms the Markov chain induced by the learned strategy and passes it to the internal model checker for verification.

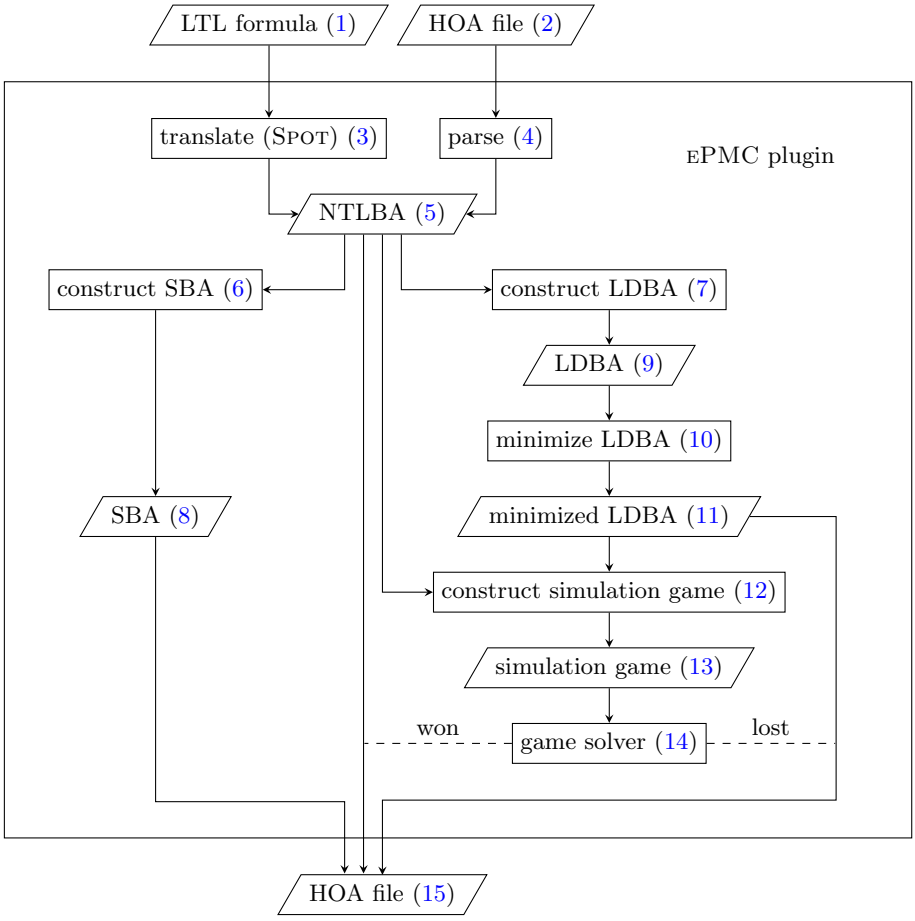


Fig. 6. Automata generation block diagram

Slim Büchi Automata Generation. For reward schemes involving LTL, the ω-regular automata translation is an important part of the design. Certain automata may be more effective for learning than others. Slim Büchi automata [13] were designed with learning considerations in mind. The translator that

produces these automata is packaged with Mungojerrie. We will now describe its design in detail for the first time.

We have implemented slim Büchi automata generation as a plugin of the probabilistic model checker ePMC [17]. The process is described in Fig. 6. The starting point is a transition-labeled Büchi automaton in HOA format [2] (2) or an LTL formula (1). In case we are given an automaton in HOA format, we parse this automaton (4) and if we are given an LTL formula, we use the tool Spot [7] to transform the formula into an automaton (3). In both cases, we end up with a transition-labeled Büchi automaton (5).

Afterwards, we have two options. The first option is to transform (6) this automaton into a slim Büchi automaton (8) [13]. These automata can then be directly composed with MDPs for model checking or used to produce rewards for learning. The other option is to construct (7) a suitable limit-deterministic Büchi automaton (SLDBA) (9). Automata of this type consist of an initial part and a final part. A nondeterministic choice only occurs when moving from the initial to the final part by an ε transition (a transition without reading a character). SLDBA can be directly composed with MDPs. However, SLDBA directly constructed from general Büchi automata are often quite large, which in turn also means that the product with MDPs would be quite large as well. Therefore, we have implemented further optimization steps. We can apply a number of algorithms to minimize (10) this automaton so as to achieve a smaller SLDBA (11). To do so, we implemented several methods:

- Subsuming the states in the final part with an empty language
- Signature-based strong bisimulation minimization in the final part
- Signature-based strong bisimulation minimization in the initial part
- Language-equivalence of states in the final part
- If we have a state s in the initial part for which we find a state s' in the final part where the language of s and s' are the same, we can remove all transitions of s and add an ε transition from s to s' instead. Afterwards, automaton states that cannot be reached anymore can be removed.

Each of these methods has a different potential for minimization as well as runtime. We therefore allow to specify which optimizations are to be used and in which order they are applied.

Once we have optimized the SLDBA, we could directly use it for later composition with an MDP. Another possibility is to prove that the original automaton is already good for MDPs. If this is the case, then it is often preferable to use the original automaton: being constructed by specialized tools such as Spot, it is often smaller than the minimized SLDBA. The original automaton is good-for-MDPs if it *simulates* the SLDBA [13]. If it does, then it is also composable with MDPs. Otherwise, it is unknown whether it is suitable for MDPs. In this case, sometimes more complex notions of simulation can be used, but existing decision procedures are too expensive to implement [36].

To show simulation, we construct (12) a simulation game, which in our case is a transition-labeled parity game (13) with 3 colors. We solve these games using (a slight variation of) the McNaughton algorithm [28]. (We are aware

that specialized algorithms for parity games with 3 colors exist [9]. However, so far the construction of the arena, not solving the game, turned out to be the bottleneck here). If the even player is winning, the simulation holds. Otherwise, more complex notions of simulation can be used, which however lead to larger parity games being constructed. In case the even player is winning for any of them, we can use the original automaton, otherwise we have to use the SLDBA. In any case, we export the result to an HOA file (15). For illustration and debugging , automata and simulation games can be exported to the GraphViz [8].

4 Case Studies

To showcase how Mungojerrie can be used to experiment with different reward schemes, we provide three case studies. In the first case study, we demonstrate how Mungojerrie can be used to compare the effectiveness of two different reward schemes on the same system. In the second case study, we consider the design space of automata, and demonstrate how Mungojerrie can be used to compare how different ω-automata change learning effectiveness. This is important for considering how to design LTL translators that produce automata that are effective for learning. In the last case study, we demonstrate how the different outputs of Mungojerrie can be used. For additional experimental results obtained using Mungojerrie, we refer readers to [11,12,14,19,39,45,23] for case studies testing ω-regular reward schemes, and [13] for the ePMC plugin. We also refer readers to [26, Fig. 3] which examined RL for scLTL properties, [6] for continuous-time MDPs, and [18], which extended Mungojerrie to test model-free reinforcement learning in branching Markov decision processes.

4.1 Comparing Reward Schemes

To demonstrate how Mungojerrie may be used to compare reward schemes, we compare the reward scheme of [11] with a modification of it that assigns a $+1$ reward on every accepting edge, as introduced in [12]. We compare these two methods on the same problem, where the learner must safely navigate two robots on a slippery gridworld to a goal. We also fix the problem parameters $\zeta = 0.99$ and $\gamma = 0.99999$, and the use of Q-learning. Since we are interested in which method will converge sooner, we fix the amount of training to be relatively low. We allow the two parameters specific to Q-learning, the learning rate α and the exploration rate ε, to be varied in order to find the optimal combination for each method. We average 10 runs for each grid point. This required 32000 runs, which took approximately 79 CPU hours (single-core) on a 2.5GHz Intel Xeon E5-2680 v3. This corresponds to an average of approximately 188000 sampled transitions per second per core, including model checking time. This sampling rate is typical of what was observed in other experiments.

Figure 7 shows the probability of satisfaction of the learned strategy as computed by the model checker of Mungojerrie. One can see that under these conditions, the reward scheme from [12] is able to consistently learn probability

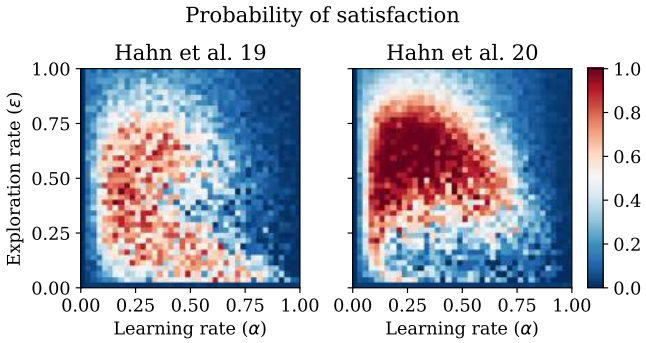


Fig. 7. Probability of satisfaction of learned strategies as computed by the model checker of Mungojerrie. 'Hahn et al. 19' refers to the translation of [11]. 'Hahn et al. 20' refers to the translation of [12] that assigns +1 reward on every accepting edge with reachability parameter ζ. Each grid point is the average of 10 runs.

1 strategies under certain parameter combinations, while [11] does not. Figure 8 shows the difference in the estimated probability of satisfaction, found by taking the value from the initial state of Q-table and renormalizing it appropriately, and the probability of satisfaction of the learned strategy computed by the model checker of Mungojerrie. One can see that the reward scheme of [11] sometimes overestimates and sometimes underestimates when it achieves a high actual probability of satisfaction under these conditions. However, on the same example, the reward scheme of [12] consistently underestimates everywhere. In summary, Mungojerrie allowed us to see that, although the reachability reward scheme of [12] may achieve higher probabilities of satisfaction sooner, it may take longer for the values in the Q-table to properly converge.

4.2 Comparing Automata

An ω-regular objective may be described by different automata, many of which may be good-for-MDPs. Mungojerrie can be used to compare the effectiveness of such automata when used in RL. Consider the two nondeterministic Büchi automata shown in Fig. 9. Both are equivalent to the LTL formula $(\mathsf{F}\,\mathsf{G}\,x) \vee (\mathsf{G}\,\mathsf{F}\,y)$, but the one on the right should be better for learning: long transient sequences of observations that satisfy $x \wedge \neg y$ may convince the agent to commit to State 1 of the left automaton too soon.

To test this conjecture, we specified a model in PRISM organized in two long chains. In one of them the agent sees many xs for a while, but eventually only sees ys. In the other chain the situation is reversed. Which chain is followed is up to chance. We then used the reward scheme from [3] with Q-learning under the default hyperparameters in Mungojerrie, $\gamma_B = 0.99$, $\gamma = 0.99999$, $\alpha = 0.1$, and $\varepsilon = 0.1$. We then trained for 20000 episodes under each automaton, and used Mungojerrie to compute the probability of satisfaction of the property at periodic

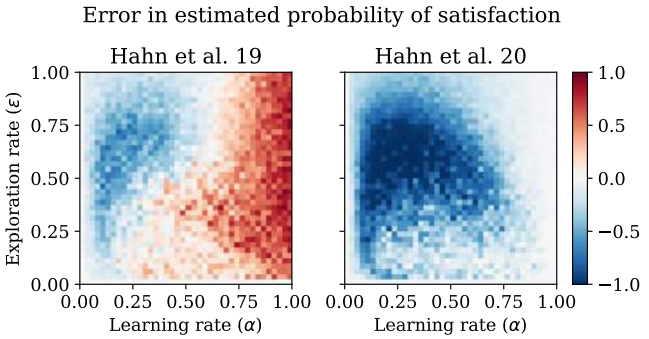


Fig. 8. Estimated probability of satisfaction of learned strategies minus the probability of satisfaction computed by the model checker of Mungojerrie. Blue indicates underestimation, while red indicates overestimation. Hahn et al. 19 refers to the translation of [11]. Hahn et al. 20 refers to the translation of [12] that assigns +1 reward on every accepting edge with reachability parameter ζ. Each grid point is the average of 10 runs.

intervals. Since learning to control the left automaton requires thorough and deep exploration, we conjectured that optimistic intialization of the Q-table [41] to the value 0.8 will improve performance. We took the average of 1000 runs for each combination.

Figure 10 shows the resulting curve. When using the LDBA without optimistic intialization, the learning agent is unable to learn the optimal strategy under these conditions. While it is worth noting that using the LDBA without optimistic initialization eventually converges to the optimal strategy with enough training, it is clear that the choice of the automaton can have a significant impact on learning performance. Therefore, the design of translations from LTL to automata has a role to play in producing effective reward schemes.

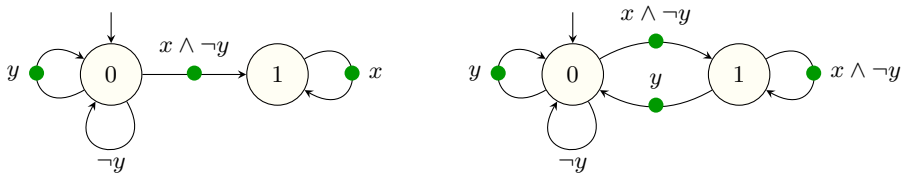


Fig. 9. Equivalent, but not equally effective, Büchi automata. "LDBA" and "Forgiving" refer to the automaton the left and right, respectively.

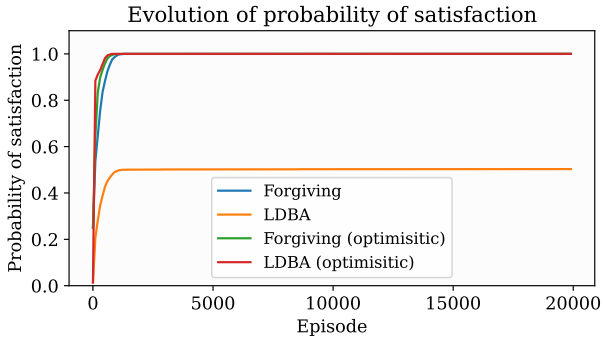


Fig. 10. Plot of the evolution of the probability of satisfaction of learned strategies as computed by the model checker of Mungojerrie. "Forgiving" and "LDBA" refer to the left and right automata in Figure 9, respectively. "(optimistic)" indicates optimistic initialization of the Q-table was used. Each curve is the average of 1000 runs.

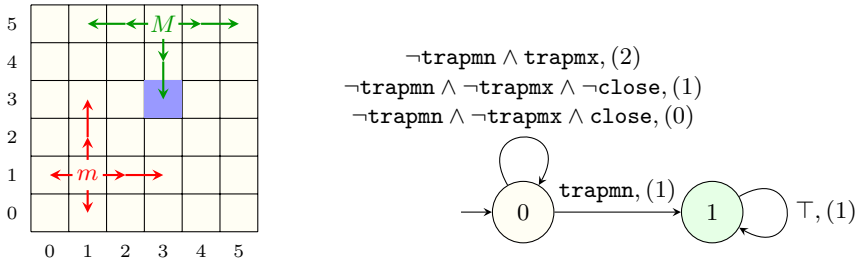


Fig. 11. A grid-world stochastic game arena (left) and a deterministic parity automaton for the objective (right).

4.3 A Game of Pursuit

Figure 11 describes a stochastic parity game of pursuit in which the Max player (M) tries to escape from the Min player (m). At each round, each player in turn chooses a direction to move. If movement in that direction is not obstructed by a wall, then the player moves either two squares or one square with equal probabilities. One square of the grid is a trap, which m must avoid at all times, but M may visit finitely many times. Player M should be at least 5 squares away from player m infinitely often. This objective is described by the LTL property $(\mathsf{F}\,\neg\texttt{trapmn}) \vee ((\mathsf{F}\,\mathsf{G}\,\neg\texttt{trapmx}) \wedge (\mathsf{G}\,\mathsf{F}\,\neg\texttt{close}))$, where `trapmn` and `trapmx` are true when m and M visit the trap square, respectively, and `close` is true when the Manhattan distance between the two players is less than 5 squares. This objective translates to the deterministic parity automaton in Fig. 11, which accepts a word if the maximum recurring priority of its run is odd.

Unlike the example of Fig. 2, inspection of the Markov chain induced by an optimal strategy and manual verification of the optimality of the learned

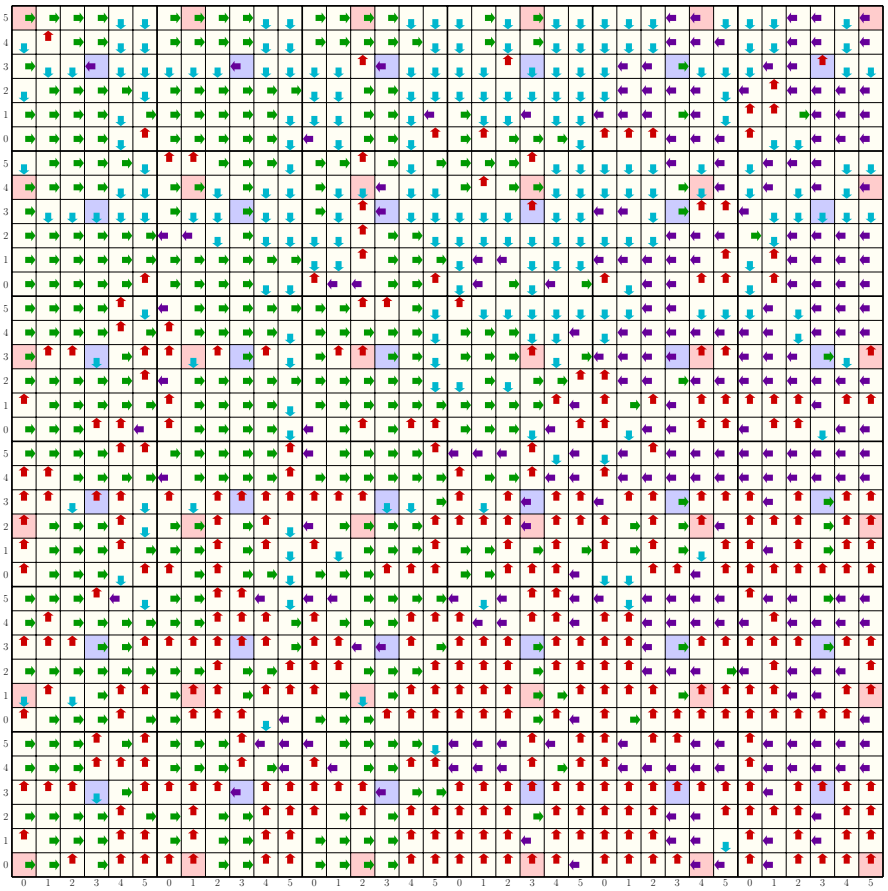


Fig. 12. Max player learned strategy for the game of Fig. 11 when the automaton is in State 0. (Any strategy will do when the automaton is in State 1.) In each 6×6 box the rose-colored square is the position of the minimizing player, while the light-blue square marks the trap.

strategy is impractical. Instead, the model checker of Mungojerrie has verified the optimality of this strategy from the intial state. For visualization, Mungojerrie can also save the strategy in CSV format. Postprocessing can then produce a graphical representation like the one of Fig. 12. The color gradient shows that, in the main, M's strategy is to move away from m.

5 Conclusion

We have introduced Mungojerrie, an extensible tool for experimenting with reward schemes for RL, with a focus on ω-regular objectives. Mungojerrie allows the specification of models in PRISM [25] and ω-automata in HOA [2]. Mul-

tiple LTL translators can be called from the tool [7,24], including the EPMC plugin introduced in [13] for the construction of slim Büchi automata. Mungojerrie includes various reward schemes [11,3,12,14,19,23,35] for ω-regular objectives and model-free RL algorithms [43,20,40,23]. Mungojerrie also includes an internal probabilistic model checker for the verification of learned strategies against ω-regular objectives, and for allowing users to verify that developed examples are as intended. The tool also comes packaged with benchmarks for ω-regular objectives in RL.

We have discussed Mungojerrie's design and demonstrated how Mungojerrie can be used to perform comparisons of reward schemes for ω-regular objectives. The source and documentation of Mungojerrie are publicly available.

References

1. de Alfaro, L.: Formal Verification of Probabilistic Systems. Ph.D. thesis, Stanford University (1998)
2. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, J., Křetínský, J., Müller, D., Parker, D., Strejček, J.: The Hanoi omega-automata format. In: Computer Aided Verification (CAV). pp. 479–486 (2015), LNCS 9206
3. Bozkurt, A.K., Wang, Y., Zavlanos, M.M., Pajic, M.: Control synthesis from linear temporal logic specifications using model-free reinforcement learning. In: 2020 IEEE International Conference on Robotics and Automation (ICRA). pp. 10349–10355 (2020). https://doi.org/10.1109/ICRA40945.2020.9196796
4. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym. CoRR **abs/1606.01540** (2016)
5. Condon, A.: The complexity of stochastic games. Inf. Comput. **96**(2), 203–224 (1992)
6. Dole, K., Gupta, A., Komp, J., Krishna, S.N., Trivedi, A.: Event-triggered and time-triggered duration calculus for model-free reinforcement learning. In: 42nd IEEE Real-Time Systems Symposium, RTSS 2021, Dortmund, Germany, December 7-10, 2021. pp. 240–252. IEEE (2021). https://doi.org/10.1109/RTSS52674.2021.00031,
7. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16). Lecture Notes in Computer Science, vol. 9938, pp. 122–129. Springer (Oct 2016)
8. Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C., Woodhull, G.: Graphviz and dynagraph - static and dynamic graph drawing tools. In: Jünger, M., Mutzel, P. (eds.) Graph Drawing Software, pp. 127–148. Springer (2004)
9. Etessami, K., Wilke, T., Schuller, A.: Fair simulation relations, parity games, and state space reduction for Büchi automata. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) Automata, Languages and Programming: 28th International Colloquium. pp. 694–707. Springer, Crete, Greece (Jul 2001), lNCS 2076
10. Hahn, E.M., Li, G., Schewe, S., Turrini, A., Zhang, L.: Lazy probabilistic model checking without determinisation. In: Concurrency Theory, (CONCUR). pp. 354–367 (2015)

11. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-regular objectives in model-free reinforcement learning. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 395–412 (2019), LNCS 11427
12. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Faithful and effective reward schemes for model-free reinforcement learning of omega-regular objectives. In: ATVA: Automated Technology for Verification and Analysis. pp. 108–124 (2020), LNCS 12302
13. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Good-for-MDPs automata for probabilistic analysis and reinforcement learning. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 306–323 (2020), LNCS 12078
14. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Model-free reinforcement learning for stochastic parity games. In: CONCUR: International Conference on Concurrency Theory. pp. 21:1–21:16 (Sep 2020), LIPIcs 171
15. Hahn, E.M., Schewe, S., Turrini, A., Zhang, L.: A simple algorithm for solving qualitative probabilistic parity games. In: Computer Aided Verification. pp. 291–311. Part II (2016), LNCS 9780
16. Hahn, E.M., Schewe, S., Turrini, A., Zhang, L.: Synthesising strategy improvement and recursive algorithms for solving 2.5 player parity games. In: Verification, Model Checking, and Abstract Interpretation. pp. 266–287 (2017)
17. Hahn, E., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: A web-based probabilistic model checker. In: International Symposium on Formal Methods. pp. 312–317 (May 2014)
18. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Model-free reinforcement learning for branching markov decision processes. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 651–673. Springer International Publishing, Cham (2021)
19. Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Model-free reinforcement learning for lexicographic omega-regular objectives. In: Formal Methods - 24th International Symposium. pp. 142–159. LNCS 13047 (2021)
20. van Hasselt, H.: Double Q-learning. In: Advances in Neural Information Processing Systems. pp. 2613–2621 (2010)
21. Henzinger, T.A., Piterman, N.: Solving games without determinization. In: 15th Conference on Computer Science Logic. pp. 394–409. Szeged, Hungary (Sep 2006), LNCS 4207
22. Irpan, A.: Deep reinforcement learning doesn't work yet. https://www.alexirpan.com/2018/02/14/rl-hard.html (2018)
23. Kazemi, M., Perez, M., Somenzi, F., Soudjani, S., Trivedi, A., Velasquez, A.: Translating omega-regular specifications to average objectives for model-free reinforcement learning. In: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems. pp. 732–741 (2022)
24. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: A library for ω-words, automata, and LTL. In: Automated Technology for Verification and Analysis, ATVA. pp. 543–550 (2018), LNCS 11138
25. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Computer Aided Verification (CAV). pp. 585–591 (Jul 2011), LNCS 6806
26. Lavaei, A., Somenzi, F., Soudjani, S., Trivedi, A., Zamani, M.: Formal controller synthesis for continuous-space mdps via model-free reinforcement learning. In: 11th ACM/IEEE International Conference on Cyber-Physical Systems,

ICCPS 2020, Sydney, Australia, April 21-25, 2020. pp. 98–107. IEEE (2020). https://doi.org/10.1109/ICCPS48487.2020.00017,
27. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems *Specification*. Springer (1991)
28. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Inf. Control. **9**(5), 521–530 (1966)
29. Mnih, V., Kavukcuoglu, K., Silver, D., et al.: Human-level control through deep reinforcement learning. Nature **518** (2015)
30. Naik, A., Shariff, R., Yasui, N., Sutton, R.S.: Discounted reinforcement learning is not an optimization problem. CoRR **abs/1910.02140** (2019), http://arxiv.org/abs/1910.02140
31. OpenAI, Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., Schneider, J., Tezak, N., Tworek, J., Welinder, P., Weng, L., Yuan, Q., Zaremba, W., Zhang, L.: Solving rubik's cube with a robot hand. arXiv preprint (2019)
32. Perrin, D., Pin, J.É.: Infinite Words: Automata, Semigroups, Logic and Games. Elsevier (2004)
33. Pnueli, A.: The temporal semantics of concurrent programs. Theoret. Comput. Science **13**, 45–60 (1981)
34. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, NY, USA (1994)
35. Sadigh, D., Kim, E., Coogan, S., Sastry, S.S., Seshia, S.A.: A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications. In: IEEE Conference on Decision and Control (CDC). pp. 1091–1096 (Dec 2014)
36. Schewe, S., Tang, Q., Zhanabekova, T.: Deciding what is good-for-mdps. CoRR **abs/2202.07629** (2022), https://arxiv.org/abs/2202.07629
37. Sickert, S., Esparza, J., Jaax, S., Křetínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Computer Aided Verification (CAV). pp. 312–332 (2016), LNCS 9780
38. Silver, D., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**, 484–489 (Jan 2016)
39. Simovec, P.: Transformation of nondeterministic büchi automata to slim automata (2021), https://is.muni.cz/th/nd15g/
40. Sutton, R.S.: Learning to predict by the method of temporal differences. Machine Learning **3**, 9–44 (1998)
41. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, second edn. (2018)
42. Wan, Y., Naik, A., Sutton, R.S.: Learning and planning in average-reward markov decision processes. In: International Conference on Machine Learning. pp. 10653–10662. PMLR (2021)
43. Watkins, C.J.C.H., Dayan, P.: Q-learning. In: Machine Learning. pp. 279–292 (1992)
44. Wiewiora, E.: Reward shaping. In: Encyclopedia of Machine Learning, pp. 863–865. Springer (2010)
45. Yang, C., Littman, M., Carbin, M.: Reinforcement learning for general ltl objectives is intractable. arXiv preprint arXiv:2111.12679 (2021)

Verification

A Formal CHERI-C Semantics for Verification

Seung Hoon Park(✉), Rekha Pai, and Tom Melham

Department of Computer Science, University of Oxford, Oxford, UK
{seunghoon.park,rekha.pai,tom.melham}@cs.ox.ac.uk

Abstract. CHERI-C extends the C programming language by adding *hardware capabilities*, ensuring a certain degree of memory safety while remaining efficient. Capabilities can also be employed for higher-level security measures, such as software compartmentalization, that have to be used correctly to achieve the desired security guarantees. As the extension changes the semantics of C, new theories and tooling are required to reason about CHERI-C code and verify correctness. In this work, we present a formal memory model that provides a memory semantics for CHERI-C programs. We present a generalised theory with rich properties suitable for verification and potentially other types of analyses. Our theory is backed by an Isabelle/HOL formalisation that also generates an OCaml executable instance of the memory model. The verified and extracted code is then used to instantiate the parametric *Gillian* program analysis framework, with which we can perform concrete execution of CHERI-C programs. The tool can run a CHERI-C test suite, demonstrating the correctness of our tool, and catch a good class of safety violations that the CHERI hardware might miss.

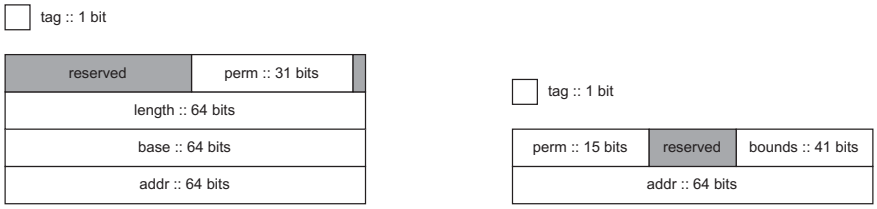
Keywords: CHERI-C · Hardware Capabilities · Memory Model · Semantics · Theorem Proving · Verification

1 Introduction

Despite having been developed more than 40 years ago, C remains a widely used programming language owing to its efficiency, portability, and suitability for low-level systems code. The language's lack of inherent memory safety, however, has been the source of many serious issues [18]. While there have been significant efforts aimed at vulnerability mitigation, memory safety issues remain widespread, with a recent study stating that 70% of security vulnerabilities are caused by memory safety issues [31].

The Capability Hardware Enhanced RISC Instructions (CHERI) project offers an alternative model that provides better memory safety [44]. Its main features include a new machine representation of C pointers called *capabilities* and extensions to existing Instruction Set Architectures (ISA) that enable the secure manipulation of capabilities. Capabilities are in essence memory addresses bound to additional safety-related metadata, such as access permissions and bounds on the memory locations that can be accessed. As the hardware performs the safety checks on capabilities, legacy C programs compiled and run

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 549–568, 2023.
https://doi.org/10.1007/978-3-031-30823-9_28



(a) CHERI-256 Capability Layout (b) CHERI-128 Capability Layout

Fig. 1: Simplified CHERI Capability Layouts

on CHERI architecture, i.e. CHERI-C code, acquire hardware-ensured spatial memory safety, while retaining efficiency. Porting code from one language to another generally requires significant efforts. But porting C codes to CHERI-C requires little, if any, changes to the original code to ensure the code runs on CHERI hardware [36,39].

In 2019, the UK announced its *Digital Security by Design* programme with £190 million of funding distributed over more than 26 research projects and 5 industrial demonstrators [6] to 'radically update the foundation of our insecure digital computing infrastructure, by demonstrating that mainstream processor technology ... can be updated to include new security technologies based on the CHERI Architecture' [5]. A cornerstone of the programme is Morello [4], a CHERI-enabled prototype developed by Arm.

Over the several years that lead to the realisation of Morello, there were several design revisions made to the hardware; examples are depicted in Fig. 1. The refined designs used methods for compression of bounds that reduced cache footprints and improved overall performance while minimising incompatibility. Morello uses a very similar design to the compressed scheme for capabilities depicted in Fig. 1b, with the overall bit-representation of the layout differing slightly. Future capability designs may possibly incorporate a different bit-representation design, provided there are improvements in performance or compatibility. Due to the ever-changing design of capability bit-representations, it seems best to have an *abstract* representation of capabilities, so that CHERI-based verification tools can remain modular.

Checking for memory safety issues of legacy C code can, of course, be achieved using existing analysis tools for C, but there are new problems that arise when such code is run on CHERI hardware. Because the pointer and memory representations are fundamentally different in a CHERI architecture, there are non-trivial differences in the semantics between C and CHERI-C.

To illustrate this point, consider the C code in Listing 1.1. This code segment performs `memcpy` twice: once from `a` to `b`, where pointers/capabilities are stored misaligned in `b`, then from `b` to `c`, where pointers/capabilities are stored correctly again in `c`. In standard C, there are no problems accessing the pointer stored in `c`. But in CHERI-C, misaligned capabilities in memory are invalidated. That means the address and meta-data of the misaligned capabilities are accessible,

but such capabilities can no longer be dereferenced [41]. While c will contain the same capability value as that of a, the capability stored in c is invalidated. Thus, the last line will trigger an 'invalid tag' exception when the code is executed on ARM Morello and other CHERI-based machines.

```
1  #include <stdlib.h>
2  #include <string.h>
3  void main(void) {
4      int *n = calloc(sizeof(int), 1);
5      int **a = malloc(sizeof(int *));
6      *a = n;
7      int **b = malloc(sizeof(int *) * 2);
8      int **c = malloc(sizeof(int *));
9      memcpy((char *) b + 1, a, sizeof(int *));
10     memcpy(c, (char *) b + 1, sizeof(int *));
11     int x = **c;
12 }
```

Listing 1.1: C code example

Of course, existing C analysis tools cannot catch these cases, as such tools are not only unaware of the changes in the semantics that capabilities bring, but also the code is not problematic in conventional C. Moreover, while CHERI ensures spatial safety by the hardware, CHERI is still incapable of catching temporal safety violations, such as Use After Free (UAF) violations. There exists work that attempt to address temporal safety [11,17,42], but they are either a software-implemented solution [42], where overall performance is inevitably affected, or ongoing work [11]. There is, therefore, a need for program analysis tools that correctly integrate the semantics of CHERI-C.

To the best of our knowledge, there is no prior work on formalising a CHERI-C memory model. The Cerberus C work [30] is primarily designed to capture pointer provenance of C programs and uses CHERI-C as a reference for pointer provenance, but the tool lacks a formal CHERI-C memory model. ESBMC is a verification tool that supports CHERI-C code [15]. But support for tagged memory does not yet exist; ESBMC would not be able to catch the 'invalid tag' exception in the code in Listing 1.1. Furthermore, ESBMC's memory model is not formally verified. Users of ESBMC must trust that the implementation of the memory model and its underlying theory are correct. SAIL formalisations for each CHERI architectures exist [3,8,9], but they only capture the low-level semantics of the architecture and not high-level C constructs such as allocation.

In this paper, we introduce a formal CHERI-C memory model that captures the memory semantics of the CHERI-C language. In Sect. 3, We formalise the memory and its operations and prove essential properties that provide correctness guarantees. We provide a rigorous logical formalisation of the CHERI-C memory model in Isabelle/HOL [32] (in Sect. 4.1) and use the code generation feature to generate a verified OCaml instance of the memory model [21]. We then show, in Sect. 4.2, the practical aspects of this work by providing the memory model to, and thereby instantiating, Gillian [20], a general, parametric verifica-

tion framework that supports concrete and symbolic execution and verification based on separation logic, backed by rich correctness properties. In Sect. 5, we demonstrate that the tool can capture the semantics of CHERI-C programs correctly. A discussion on the existing works can be found in Sect. 6 while Sect. 7 concludes this paper mentioning possible future directions. We first start with an introduction to the CHERI architecture.

2 CHERI

CHERI extends a conventional ISA by introducing *capabilities* which are essentially pointers that come along with metadata to restrict memory access. The ISA now has additional hardware instructions and exceptions that operate over capabilities. Register sets are extended to include capability registers, instructions are added that reference the capability registers, and custom hardware exceptions are added to block operations that would violate memory safety. Designs of CHERI capabilities have refined over the past several years and have been incorporated in several existing architectures, such as MIPS and RISC-V [40]. All CHERI-extended ISAs have been formally defined using the SAIL specification language, in which the logic of machine instructions and memory layout have been defined formally in a first-order language [13].

Regardless of the layout, CHERI capabilities include three important types of high-level information, in addition to a 64-bit address:

- **Permissions.** Permissions state what kind of operations a capability can perform. Loading from memory and storing to memory are examples of permissions a capability may possess.
- **Bounds.** Bounds stipulate the memory region that the address part of a capability can reference. The lower bound stipulates the lowest address that a capability may access, and the upper bound stipulates the highest address.
- **Tag.** Stored separately from the other components of a capability, the tag states the validity of the capability it is attached to. Capabilities with invalid tags can hold data but cannot be dereferenced. Attempts to forge capabilities out of thin air result in a tag-invalidated capability.

Fig. 1a show a 256-bit representation of a capability, which was one of the earlier designs. The lower and upper bounds are represented using the base and length fields. Here, the lower bound is the address stated by the base field, and the upper bound is the address in the base field plus the length field. Permissions and other metadata are stored in the remaining fields as a bit vector. The capability's tag bit exists separately from the capability. Tag bits are, in practice, stored separately from the main memory where capabilities reside, so users cannot manipulate the tag bits of capabilities stored in memory. Furthermore, overwriting capabilities stored in memory with non-capability values invalidates their tag bits, which ensures capabilities cannot be forged out of thin air.

This representation, in theory, exercises a high level of compatibility with existing C code. But performance, particularly with regards to caching, is reduced

due to the size of the capability representation [43]. Refined designs ultimately resulted in a capability that utilises a floating-point-based lossy compression technique on the bounds [43], such as the one depicted in Fig. 1b. In many cases, the upper bits of the address fields are most likely to overlap with those of the lower and upper bounds. Knowing this, bounds can be compressed by having the upper bits of their fields depend on that of the address, which means only the lower bits need to be stored.

The lossy compression of bounds may result in some incompatibility. Bounds may no longer be represented exactly, and changes in the address field may result in an unintentional change in the bounds. Nonetheless, such representations give an acceptable level of compatibility, provided aggressive pointer arithmetic optimisations are avoided. The Morello processor incorporates a similar compression-based design in its architecture, though sizes of each field differ [12].

The added capability-aware instructions operate over capabilities. Conventional load and store operations are extended to first check that the tag, permissions, and bounds of the capability are all valid. Violations result in triggering a capability-related hardware exception. There are additional operations to access or change the tag, permissions, and bounds. To ensure spatial memory safety, these operations can, at most, make the conditions for execution more restrictive; they cannot grant that which was not previously available. For instance, one cannot lower the lower bound of a capability to access a region that was inaccessible before, or grant a store permission that was unset beforehand. Because of how tags work for capabilities stored in memory, one cannot grant capabilities larger bounds or more permissions by manipulating the memory—attempting this results in tag invalidation.

Library support for CHERI has grown over the past few years. In particular, a software stack for CHERI-C that utilises a custom Clang compiler now exists [41]. Users can compile their program either in 'purecap' mode, where all pointers in programs are replaced with capabilities, or in 'hybrid' mode, where both pointers and capabilities co-exist within the program. Because operations that change the fields of a capability does not generally exist in standard C, Clang incorporates additional CHERI libraries of operations that users may use to access or mutate capabilities.

3 CHERI-C Memory Model

Incorporating hardware-enabled spatial safety requires significant changes to the C memory model. Pointer designs must be extended to incorporate bounds, metadata, and the out-of-band tag bit. The memory, i.e. heap, must also be able to distinguish the main memory and the tagged memory. Operations with respect to the heap must also be defined such that tag preservation and invalidation are incorporated appropriately.

In this section, we provide a generalised theory for the CHERI-C memory model. We identify the type and value system used by the memory model. We then define the heap and the core memory operations. Finally, we state some

essential properties of the heap and the operations that (1) characterises the semantics and (2) states what types of verification or analyses could be supported. We make the assumption that we work on a 'purecap' environment, where all pointers have been replaced with capabilities.

3.1 Design

The CHERI-C memory model is inspired by that of CompCert [26]. The beauty of CompCert is that it is a verified C compiler. The internal components, which include the block-offset based memory model, are formalised in a theorem prover, with many of its essential properties verified. Using CompCert's memory model as a basis, we design the CHERI-C memory model by providing extensions to ensure the modelling of correct semantics and the capture of safety violations:

- **Capability Values.** In addition to the standard primitive types, we incorporate abstract capabilities as values. We also incorporate capability fragments to provide semantics to higher-level memory actions like `memcpy`, which should preserve tags if copied correctly and invalidate otherwise [41].
- **Extended Operations.** Basic memory actions such as `load` and `store` now work on capabilities and will trigger the correct capability-related exception when required.
- **Tagged Memory.** Tags in memory are stored separately from the main heap, as could be seen by the formal CHERI-MIPS SAIL model [9]. So we provide a separate mapping for tagged memory for storing capability tags.
- **Freed Regions.** The standard CompCert memory model can mark which memory regions are valid but lacks the ability to distinguish which regions are marked as 'Freed'. We incorporate freed regions as a means to catch temporal safety violations.

3.2 Type and Value System

Figure 2 shows the formalisation of CHERI-C types and values. Types τ are analogous to chunks in CompCert terms. Types comprise primitive types (e.g. $\mathrm{U8}_\tau$,

$$
\begin{array}{lcl}
\tau & \triangleq & U8_\tau \mid S8_\tau \mid ... \mid U64_\tau \mid S64_\tau \mid Cap_\tau \\
MCap & \triangleq & \mathcal{B} \times \mathbb{Z} \times md \\
Cap & \triangleq & MCap \times \mathbb{B} \\
\mathcal{V}_\mathcal{C} & \triangleq & U8_\mathcal{V} \ :: \ 8 \ bits \mid ... \\
 & & \mid S64_\mathcal{V} \ :: \ 64 \ sbits \\
 & & \mid Cap_\mathcal{V} \ :: \ Cap \\
 & & \mid CapF_\mathcal{V} \ :: \ Cap \times \mathbb{N} \\
 & & \mid Undef \\
\mathcal{V}_\mathcal{M} & \triangleq & Byte \ :: \ 8 \ bits \\
 & & \mid MCapF \ :: \ MCap \times \mathbb{N}
\end{array}
$$

Fig. 2: CHERI-C Types and Values

$S64_\tau$, etc.) and a capability type Cap_τ. We define a function $|\cdot| : \tau \to \mathbb{N}$ that returns, in terms of bytes, the size of the type. For Cap_τ, the value is not fixed but requires that it must be divisible by 16. This requirement allows capabilities with 128- and 256-bit representations to have a valid size.

$MCap$ represents a *memory capability* value and is represented as a tuple (b, i, m), which comprises the block identifier $b \in \mathcal{B}$, offset $i \in \mathbb{Z}$, and metadata $m \in md$, where md represents the bounds and permissions. Here, $\mathcal{B}$ must be a countable set. Offsets are represented as integers, as CHERI allows out-of-bounds addresses, where the address may be lower than the lower bound. Because capabilities stored in memory have their tag bit stored elsewhere, we make the distinction between memory capabilities and *tagged capabilities*, Cap, which is a capability $((b, i, m), t)$ that contains the tag bit $t \in \mathbb{B}$.

Unlike those of CompCert, CHERI-C values $\mathcal{V}_\mathcal{C}$ are given type distinctions to ensure: (1) types can be inferred directly, and (2) they contain the correct values at all times. From a practical standpoint, this ensures that the proof of correctness of memory operations can be simplified, and bounded arithmetic operations can be implemented correctly. Capability values $Cap_\mathcal{V}$ and capability fragment values $CapF_\mathcal{V}$ also exist as values. Provided some capability value $C \in Cap_\mathcal{V}$, capability fragment values $C_n \in CapF_\mathcal{V}$ correspond to the n-th byte of the capability C. For both cases, instead of fixing their representation concretely, we represent them abstractly using a tuple. This representation ensures that conversion to a compressed representation could be achieved when needed while avoiding the need to fix to one particular bit representation. Furthermore, this approach provides a reasonable way to correctly define `memcpy`, where capability tags must be preserved if possible. While capability fragments are extended structures of capabilities, operations that can be performed on capability fragments are limited. Finally, we have *Undef*, which represents invalid values. These values may appear when, for example, the user calls `malloc` and immediately tries to load the undefined contents. The idea behind incorporating capability fragments values is heavily inspired by the work from [25].

Because values are given a type distinction, identifying the types of values is straightforward. For capability fragments, we have two choices: they may either be a $U8_\tau$ or $S8_\tau$ type. Capability fragments are essentially bytes, so operations over capability fragments can be treated as if they were a $U8_\tau$ or $S8_\tau$ type. Since *Undef* does not correspond to a valid value, it is not assigned a type.

$$
\begin{array}{lcl}
\text{CapErr} & \triangleq & \text{TagViolation} \mid \text{PermitLoadViolation} \mid \ldots \\
\text{LogicErr} & \triangleq & \text{UseAfterFree} \mid \text{MissingResource} \mid \ldots \\
\text{Err} & \triangleq & \text{CapErr} \mid \text{LogicErr} \\
\mathcal{R}\ \rho & \triangleq & \mathcal{S}ucc\ \rho \\
 & & \mid \mathcal{F}ail\ \text{Err}
\end{array}
$$

Fig. 3: CHERI-C Errors

Memory operations, such as `load` and `store`, are defined so that, upon failure, the operation returns the type of error that lead to the failure. In general, partial functions, or function using the option type, can model function failure but cannot state what caused the failure. As such, the operations use the return type $\mathcal{R}\ \rho$, where ρ is a generic return type. For CHERI-C, we make the distinction between errors caused by capabilities, denoted by CapErr, and errors caused by the language, denoted by LogicErr. Figure 3 depicts the formalised Errors system used by the memory model.

3.3 Memory

We now formalise the memory. We use CompCert's approach of using a union type $\mathcal{V}_{\mathcal{M}}$ that can represent either a byte or a byte fragment of a memory capability. Then it is possible to create a memory mapping $\mathbb{N} \rightharpoonup \mathcal{V}_{\mathcal{M}}$.[1] We also create a separate mapping of type $\mathbb{N} \rightharpoonup \mathbb{B}$ for tagged memory. When the user attempts to store a capability, it will be converted into a memory capability and then stored in the memory mapping. Separately, the tag bit will be stored in the tagged memory. When the tag bit is stored, adjustments are made to ensure tags are only stored in capability-size-aligned offsets.

To ensure we can catch temporal safety violations, we need to be able to make distinctions between blocks that are freed and blocks that are valid. One way to encode this is as follows: a block b may point to either a freed location (i.e. $b \mapsto \varnothing$), or point to the pair of maps we defined earlier. The idea is that if a block identifier points to a freed block, attempts to load such a block will trigger a 'Use After Free' violation and would otherwise point to a valid mapping pair. Ultimately, the heap has the following form:

$$\mathcal{H} : \mathcal{B} \rightharpoonup ((\mathbb{N} \rightharpoonup \mathcal{V}_{\mathcal{M}}) \times (\mathbb{N} \rightharpoonup \mathbb{B}))_{\varnothing}$$

3.4 Operations

We define the core memory operations, or *actions*, of the memory model. We use the same result type $\mathcal{R}$ given in Fig. 3 instead of using a partial function to give the type of error, should the operation fail.

The memory actions $A_{\mathcal{C}} = \{\texttt{alloc}, \texttt{free}, \texttt{load}, \texttt{store}\}$ are given below with their respective signatures:

- `alloc` : $\mathcal{H} \to \mathbb{N} \to \mathcal{R}\ (\mathcal{H} \times Cap)$
- `free` : $\mathcal{H} \to Cap \to \mathcal{R}\ (\mathcal{H} \times Cap)$
- `load` : $\mathcal{H} \to Cap \to \tau \to \mathcal{R}\ (\mathcal{V}_{\mathcal{C}})$
- `store` : $\mathcal{H} \to Cap \to \mathcal{V}_{\mathcal{C}} \to \mathcal{R}\ (\mathcal{H})$

[1] The notation $\rightharpoonup$ denotes a partial map. Offsets in heaps are $\mathbb{N}$, whereas offsets stored in capabilities are $\mathbb{Z}$. Operations check whether the offsets are in bounds, which requires offsets to be non-negative. This means valid offset values can be converted from $\mathbb{Z}$ to $\mathbb{N}$ without issues.

The function `alloc` $\mu\ n = \mathcal{S}ucc\ (\mu', c)$ takes a heap μ and size n input and produces a fresh capability c and the updated heap μ' as output. The bounds of c are determined by n. In the case of compressed capabilities, a sufficiently large n *may* result in the upper bound being larger than what was requested. The capability c is also given the appropriate permissions and a valid tag bit. Like that of CompCert, `alloc` is designed to never fail, provided that the countable set $\mathcal{B}$ has infinite elements.

The function `free` $\mu\ c = \mathcal{S}ucc\ (\mu', c')$ takes a heap μ and capability $c = ((b, i, m), t)$ as input. Upon success, the operation will return the updated heap, where we now have $b \mapsto \varnothing$. The capability c' is also updated such that the tag bit of c is invalidated. This conforms to the CHERI-C design stated in [41]. We note that c should also be a valid capability, that is—at the very least—the tag bit should be set, and the offset should be within the capability bounds. The function `free` may fail if the block is invalid or already freed, even if the capability itself was valid. In such case, `free` returns a logical error.

The function `load` $\mu\ c\ t\ = \mathcal{S}ucc\ v$ takes a heap μ, capability c and type t as input, where t is the type the user wants to load. Upon success, the operation will return the value v from the memory, where v has the corresponding type t.[2] Before `load` attempts to access the block provided by c, it first checks that c has sufficient permissions to load. We use the CHERI-MIPS SAIL implementation of the CL[C] instruction [40] for the capability checks, implementing the extra checks provided that $t = Cap_\tau$. Once the capability checks are done, the operation attempts to access the blocks and the mappings, failing and returning the appropriate logical error if they do not exist.

When accessing both the main memory and tagged memory, there are a number of cases to consider. When loading primitive values, it is important that the region about to be loaded is all of *Byte* and not of *MCapF* type. Thus, before loading the values, we check whether the contiguous region in memory are all of *Byte* type. If this is not the case, `load` will return *Undef*. For capability fragments, the cell in memory has to be an *MCapF*. Finally for capabilities, not only do the contiguous cells have to be of *MCapF* type, but (1) they must have the same memory capability value, and (2) the fragment values must all be a sequence forming $\{0, 1, ..., |Cap_\tau| - 1\}$. The idea is that even if the contiguous cells have the same memory capability values, they do not form a valid capability if the fragments are not stored in order. After all the checks, the tagged memory will be accessed, where the tag value is retrieved.[3] The loaded memory capability and tag bit are then combined to form a tagged capability, which `load` returns.

The function `store` $\mu\ c\ v\ = \mathcal{S}ucc\ \mu'$ takes a heap μ, capability c, and value v. Upon success, the operations will return the updated heap μ'. Like `load`, `store` performs the necessary capability checks based on CHERI-MIPS' CS[C] instruction and attempts to access the blocks and mappings afterwards, returning the appropriate exception upon failure. For storing primitive values and capability

[2] For capability fragments, the corresponding type may be either $U8_\tau$ or $S8_\tau$.

[3] The tagged memory does not need to be accessed if c does not have a capability load permission. In such case, the loaded capability will have an invalidated tag.

fragment values, the main memory mapping will simply be updated to contain the values, and the associated tagged memories will be invalidated. For primitive values that are not bytes, the values will be converted into a sequence of bytes, where each byte in the list will be stored contiguously in memory. For a capability fragment value, it will be stored in the cell as an *MCapF* type, where the tag value of the fragment will be stripped when storing in memory. Finally, for capability values, the value will be split into a list comprising $|Cap_\tau| - 1$ memory capability fragments, with the fragment value forming a sequence $\{0, 1, ..|Cap_\tau| - 1\}$, and a tag bit. The main memory will store the list of memory fragments contiguously, and the tagged memory will store the tag value in the corresponding capability-aligned tagged memory.

3.5 Properties

In the previous section, we have articulated a formal CHERI-C memory model, explaining how the heap is structured and how the operations are defined. It is essential that the formalisation we provided is correct and is also suitable for verification or other types of analyses. In this section, we first discuss the properties of the memory. We then discuss the properties of the operations themselves, primarily concerned with correctness.

When we observe the memory, it is important that we always work with a valid one, i.e. the memory is *well-formed*. In our formalisation, we require that all tags in the tagged memory are stored in a capability-aligned location. The well-formedness relation $\mathcal{W}_f^{\mathcal{C}}$ is defined as follows:

$$\mathcal{W}_f^{\mathcal{C}}(\mu) \equiv \forall b \in dom(\mu).\ b \mapsto (c, t) \longrightarrow \forall x \in dom(t).\ x \bmod |Cap_\tau| = 0$$

The well-formedness property must hold when the heap is initialised and when memory operations mutate the heap. That is, provided μ_0 is the initialised heap where all mappings are empty, $\alpha \in A_{\mathcal{C}}$ is a memory action, v are the arguments of the memory operation α and μ' is one of the return values denoting the updated heap, we have the following properties:

$$\mathcal{W}_f^{\mathcal{C}}(\mu_0)$$

$$\mathcal{W}_f^{\mathcal{C}}(\mu) \Longrightarrow \alpha\ \mu\ v = \mathcal{S}ucc\ \mu' \Longrightarrow \mathcal{W}_f^{\mathcal{C}}(\mu')$$

The two properties above ensure that the heap is well-formed throughout the execution of the CHERI-C program.

For the correctness of the operations, we primarily consider soundness and completeness:

- If the inputs are valid for operation $\alpha \in A_{\mathcal{C}}$ then the action should succeed.
- If the action α succeeds, the inputs provided to the operations are valid.
- If the inputs are invalid for the operation α, then the action should fail and return the correct error.

The first and second points are simple soundness and completeness properties. The third point is important in that the input may be problematic in many ways. For example, the NULL capability has an invalid tag bit, invalid bounds, and no permissions. The function `load` will fail if provided with the NULL capability, as it violates many of the checks. Because the SAIL specification states that tags are always checked first, the error must be a TagViolation type.

Next, we need to ensure successive operations yield the desired result. The primary properties to consider are the *good variable* laws [26]; examples of properties encoding this law include *load after allocation*, *load after free*, and *load after store*. It is worth mentioning there are some caveats. For example, the *load after store* case no longer guarantees that you will retrieve the same value you stored, unlike CompCert's load after store property in [26], since the value that was stored and to be loaded again could have been either a capability or capability fragment. In such cases, the tag bit may become invalidated due to insufficient permissions on the capability, or because storing capability fragments resulted in the tagged memory being cleared. The solution is to divide the general property into a primitive value case and a capability-related value case. Ultimately, the idea is to prove that the loaded value is *correct* rather than exact, i.e. capability-related values when loaded with have the correct tag value.

Finally, we have properties suitable for verification. We note that the memory $\mathcal{H}$ can be instantiated as a separation algebra by providing the partial commutative monoid (PCM) $(\mathcal{H}, \uplus, \mu_0)$, where $\uplus$ is the disjoint union of two heaps and μ_0 is the empty initialised heap. For tools that rely on using partial memories, it is also imperative to show that the well-formedness property is compatible with memory composition:

$$\mathcal{W}_f^{\mathcal{C}}(\mu_1 \uplus \mu_2) \Longrightarrow \mathcal{W}_f^{\mathcal{C}}(\mu_1) \wedge \mathcal{W}_f^{\mathcal{C}}(\mu_2)$$

We also note that the current heap design keeps track of *negative* resources [28], which may potentially be useful for incorrectness logic based verification [33].

4 Application

The overall memory model provided in Sect. 3 has been designed to be applicable for verification tools. In this section, we explain how we use the theory provided above to create a verified, executable instance of the memory model. We then explain how this executable model can be used to instantiate a tool called Gillian [20]. Using the instantiated tool, we demonstrate the concrete execution of CHERI-C programs with the desired behaviour.

4.1 Isabelle/HOL

Isabelle/HOL is an interactive theorem prover based on classical Higher Order Logic (HOL) [32]. We use Isabelle/HOL to formalise the entirety of the CHERI-C memory model discussed in Sect. 3. Types, values, heap structure,

etc. were implemented, memory operations were defined, and properties relating to the heap and the operations were proven. Memory capabilities, tagged capabilities, and capability fragments were represented using records, a form of tuple with named fields. For code generation, we instantiated the block type $\mathcal{B}$ to be $\mathbb{Z}$. For showing that $\mathcal{H}$ is an instance of a separation algebra, we use the `cancellative_sep_algebra` class [23] and prove that the heap model is an instance. This proof ultimately shows that $\mathcal{H}$ forms a PCM. Proving that well-formedness is compatible with memory composition is stated slightly differently. The `cancellative_sep_algebra` class takes in a total operator $\cdot_t$ instead of a partial one and requires a 'separation disjunction' binary operator $\#$, which states disjointedness. Ultimately, the compatibility property can be given as:

$$\mu_1 \# \mu_2 \Longrightarrow \mathcal{W}_f^{\mathcal{C}}(\mu_1 \cdot_t \mu_2) \Longrightarrow \mathcal{W}_f^{\mathcal{C}}(\mu_1) \wedge \mathcal{W}_f^{\mathcal{C}}(\mu_2)$$

For partial mappings of the form $A \rightharpoonup B$, we use Isabelle/HOL's finite mapping type `('a,'b)mapping` [22]. To ensure we obtain an OCaml executable instance of the memory model, we use the Containers framework [27], which generates a Red-Black Tree mapping provided the abstract mapping in Isabelle/HOL. All definitions in Isabelle were either defined to be code-generatable to begin with (i.e. definitions should not comprise quantifiers or non-constructive constants like the Hilbert choice operation $SOME$), or code equations were provided and proven to ensure a sound code generation [21]. For bounded machine words, which is required for formalising the primitive values, we use Isabelle/HOL's word type `'a word`, where `'a` states the length of the word [14]. Types like `'a word`, `nat`, `int` and `string` were also transformed to use OCaml's Zarith and native string library for efficiency [21].

4.2 Gillian

Gillian is a high-level analysis framework, theoretically capable of analysing a wide range of languages. The framework allows concrete and symbolic execution, verification based on Separation Logic, and bi-abduction [28]. The crux of the framework lies in its parametricity, where the tool can be instantiated by simply providing a compiler front end and OCaml-based memory models of the language. So far, CompCert C and JavaScript have both been instantiated for Gillian, giving birth to Gillian-C and Gillian-JS.

The underlying theoretical foundation of Gillian has its essential correctness properties like soundness and completeness already proven [20, 29]. Thus, users who instantiate the tool only need to prove the correctness of the implementation of their compiler and memory models to ensure the correctness of the entire tool. From the perspective of someone trying to instantiate Gillian with their compiler and memory models, it is essential to understand the underlying intermediate language GIL and the overall memory model interface used by Gillian.

GIL GIL is the GOTO-based Intermediate Language used by Gillian which is used for all types of analyses the tool supports. For concrete execution, GIL

supports basic GOTO constructs and assertions. For symbolic execution, the GIL grammar is extended to support path cutting, i.e. assumptions, and generation of symbolic variables. For separation logic based verification, the GIL grammar is further extended to support core predicates and user-defined predicates [28] that can be utilised to form separation logic based assertions. Furthermore, function specifications in the Hoare-triple form $\{P\}f(\bar{x})\{Q\}$ can be provided, where P and Q are separation logic based assertions.

Note that Gillian uses a value set $\mathcal{V}$ which differs from that used in the CHERI-C memory model. As we are only interested in the values used in the CHERI-C memory model, it is possible to implement a thin conversion layer between the two value systems. We note that a list of GIL values also constitutes a GIL value, so arguments for functions can be expressed as a single GIL value. This is important when understanding the memory model layout of Gillian.

Memory Model Memory Models in Gillian have a specific definition and have properties that state what kind of analysis is supported. Proving that the provided memory models satisfy certain properties is essential in understanding what the instantiated tool supports.

Gillian differentiates between concrete and symbolic memory models, which are used for concrete and symbolic execution, respectively. As we are concerned with concrete execution, we will consider only concrete memory models here.

At the highest level, there are two kinds of memory model properties: *executional* and *compositional*. The *executional* memory model states properties a memory model must have for whole-program execution, and the *compositional* memory model states properties a memory model must have for separation logic based symbolic verification. Each paper in the Gillian literature states slightly different definitions for the memory models [20,28,29,37]—in Definitions 1 and 2 below, we present unified, consistent definitions for each of the memory model properties. We ignore contexts, as there exists only one context in concrete memories, which is the GIL boolean value `true`.

Definition 1. *(Execution Memory Model). Given the set of GIL values $\mathcal{V}$ and an action set A, an execution memory model $M(\mathcal{V}, A) \triangleq (|M|, \mathcal{W}_f, \underline{ea})$ comprises:*

1. *a set of memories $|M| \ni \mu$*
2. *a well-formedness relation $\mathcal{W}_f \subseteq |M|$, with $\mathcal{W}_f(\mu)$ denoting μ is well-formed*
3. *the action execution function $\underline{ea} : A \to |M| \to \mathcal{V} \to \mathcal{R}\ (|M| \times \mathcal{V})$*

Definition 2. *(Compositional Memory Model). Given the set of GIL values V and core predicate set Γ, a compositional memory model, $M(V, A_\Gamma) \triangleq (|M|, \mathcal{W}_f, \underline{ea}_\Gamma)$ comprises:*

1. *a partial commutative monoid (PCM) $(|M|, \cdot, 0)$*
2. *A well-formedness relation $\mathcal{W}_f \subseteq |M|$ with the following property:*

$$\mathcal{W}_f(\mu_1 \cdot \mu_2) \implies \mathcal{W}_f(\mu_1) \wedge \mathcal{W}_f(\mu_2)$$

3. the predicate action execution function $\underline{ea}_\Gamma : A_\Gamma \to |M| \to V \rightharpoonup \mathcal{R}\ (|M| \times V)$

First, we note that for concrete execution, Gillian also uses the return type $\mathcal{R}$ in the action execution function $\underline{ea}$.[4] For $\mathcal{W}_f$ defined in Definition 1, the main properties that must be satisfied are Properties 3.1, 3.2, and 3.6 in [29].

The PCM requirement is required to show that the heap forms a separation algebra [16]. $\mathcal{W}_f$ is extended to state that memory composition must also be well-formed. Finally, the predicate action execution function $\underline{ea}_\Gamma$ provides a way to frame on and off parts of the memory, though they are not required for concrete execution as they are not part of the GIL concrete execution grammar.

Using the CHERI-C memory model we defined earlier, we can show that our model conforms to both Definitions 1 and 2. Let $A_\mathcal{C}$ be the set of memory actions, $\mathcal{H}$ be the memory, $\underline{ea}_\mathcal{C}$ be the action execution function of the CHERI-C memory model, and $\mathcal{W}_f^\mathcal{C}$ be the well-formedness relation. Then we observe that $(\mathcal{H}, \mathcal{W}_f^\mathcal{C}, \underline{ea}_\mathcal{C})$ forms an execution memory model. We note that Properties 3.1 and 3.2 in [29] are satisfied, and Property 3.6 is trivial in that operations that return errors do not return an updated heap. We also note that the memory model also conforms to a compositional memory model, as we have the PCM $(\mathcal{H}, \uplus, \mu_0)$ along with the well-formedness property being composition-compatible. The predicate action execution function is not required to be given, as the concrete execution of Gillian does not utilise this feature.

4.3 Compiler

We implemented a CHERI-C to GIL compiler by utilising ESBMC's GOTO language. The idea is that ESBMC uses its own intermediate representation for bounded model checking, which is the GOTO language. CHERI-enabled ESBMC uses Clang as a front end to generate the GOTO language. In our case we can build a GOTO to GIL compiler instead of building a CHERI-C compiler from scratch. The GOTO language is very similar to GIL in that they are both goto-based languages and uses single static assignment. For most parts, the compilation process is straightforward. As ESBMC's GOTO language is typed while the CHERI-C memory model is untyped—untyped in the sense that the memory model does not support user-defined types like `structs`—we make sure that capability arithmetic and casts are applied correctly by inferring the sizes of the user-defined types.

5 Experimental Results

In Sect. 4, we have provided a way to instantiate the Gillian tool, where we obtain a concrete CHERI-C model using Isabelle/HOL and a CHERI-C to GIL

[4] In the Gillian literature, it is stated that $\mathcal{R}$ can return both a return value and an error. The OCaml implementation of Gillian slightly differs from this and is more similar to $\mathcal{R}$ used for the CHERI-C memory model.

compiler that utilises ESBMC's GOTO language. Our framework can demonstrate that higher-level memory actions—such as `memcpy()`, which preserves tags when applicable—can be implemented. Furthermore, we can run concrete instances of programs that use `memcpy()` to show they emit the expected behaviour. This also means the tool can catch the TagViolation exception that is triggered in Listing 1.1. Our tool also allows capability-related functions defined in `cheriintrin.h` and `cheri.h`, to be usable, i.e. it is possible to call operations such as `cheri_tag_get()` and `cheri_tag_clear()`.

Filename	GC	GCC	AM	BMC
`buffer_overflow.c`	✓	✓	✓	✓
`dangling_ptr.c`	✓	✓	×	✓
`double_free.c`	✓	✓	×	✓
`invalid_free.c`	×[5]	✓	✓	✓
`misaligned_ptr.c`	✓	✓	✓	×
`listing_1.c`	×	✓	✓	×

Table 1: Violation detection

Filename	Time(s)
`libc_malloc.c`	**8.585**
`libc_memcpy.c`	1.698
`libc_memmove.c`	0.318
`libc_string.c`	0.315

Table 2: GCC runtime performance

Table 1 shows a list of safety violations that Gillian-C, our tool, the ARM Morello hardware, and CHERI-ESBMC—labelled as GC, GCC, AM, and BMC, respectively—all catch. We observe that Morello fails to catch temporal safety violations such as dangling pointers and double frees. For the invalid free case, where we attempt to free a pointer not produced by `malloc`, we discovered a bug in the Gillian-C tool that fails to catch this violation.[5] Gillian-C does not return any errors for the program in Listing 1.1, which is to be expected, as this is not problematic for conventional C. Finally, we observe that CHERI-ESBMC fails to catch the last two violations that relating to tag invalidation.

Table 2 shows the runtime performance of running the CHERI-C library test suites, based on the Clang CHERI-C test suite [1]. Tests were conducted on a machine running Fedora 34 on an 11$^{\text{th}}$ Gen Intel Core i7-1185G7 CPU with 31.1 GB RAM, with trace logging enabled. We note that when the test cases were executed on Morello without any modifications to the code, all of the tests terminated instantaneously without any issues. In the `libc_malloc.c` test case, we reduced the scope of the test[6] to ensure the tool terminates within a reasonable time, though the performance can be drastically improved by turning logging off, e.g. the `libc_malloc.c` case would only take 0.686 seconds. For the remaining tests, we made modifications to the code to ensure the compiler can correctly produce the GIL code, and we made sure to preserve all the edge cases covered by the original tests. For example, in `libc_memcpy.c` we made sure to test all cases where both `src` and `dst` capabilities were aligned and misaligned in the beginning and the end, which affected tag preservation. We observed that no assertions were violated, and we also observed that the same

[5] The bug has since been fixed after a discussion with the developers [7].

[6] In particular, we reduced `max` from the `libc_malloc.c` case in [1] from 20 to 9.

code when run in Morello also resulted in no assertion violations, demonstrating a faithful implementation of CHERI-C semantics.

6 Related Work

The CompCert C memory model [26], CH_2O memory model [24], and Tuch's C memory model [38] are C memory models formalised in a theorem prover, each focusing on different aspects of verification. Our model mostly draws inspiration from these models, extending such work to support CHERI-C programs.

VCC, which internally uses the typed C memory model [19], and CHERI-ESBMC [15] are designed with automated verification of C programs via symbolic execution in mind—in particular, CHERI-ESBMC supports hybrid settings and compressed capabilities in addition to purecap settings and uncompressed capabilities. Both tools rely on a memory model that is not formally verified, so the tools have components that must be trusted.

7 Conclusion and Future Work

We have provided a formal CHERI-C memory model and demonstrated its utility for verification. We formalised the entire theory in Isabelle/HOL and generated an executable instance of the memory model, which was then used to instantiate a CHERI-C tool. The result lead to a concrete execution tool that is robust in terms of the properties that are guaranteed both by the tool and by the memory model. We demonstrated its practicality by running CHERI-C based test suites, capturing memory safety violations, and comparing the results with actual CHERI hardware—namely the physical Morello processor.

Currently there are a number of limitations provided by the memory model. Capability arithmetic is limited only to addition and subtraction, but the heap can be extended to incorporate mappings from blocks to physical addresses and vice versa. This provides a way to extend capability arithmetic. While the theory incorporates abstract capabilities, compression is still under work. We believe, however, that the abstract design itself does not need to change. It may be possible to utilise the compression/decompression work to convert between the two forms [2] when needed whilst retaining our design for the operations.

This theory serves as a starting point for much potential future work. A compositional symbolic memory model can be built from this design to enable symbolic execution and verification in Gillian. As we have already proven the core properties, proving the remaining properties for the extended model will allow automated separation logic based verification of CHERI-C programs.

Acknowledgements We are very grateful to the Gillian team, in particular, Sacha-Élie Ayoun, for providing assistance with instantiating the Gillian tool. We also thank Fedor Shmarov and Franz Brauße for providing assistance with building and modifying the ESBMC tool. This work was funded by the UKRI programme on Digital Security by Design (Ref. EP/V000225/1, SCorCH [10]).

Data-Availability Statement The Isabelle/HOL formalisation of the CHERI-C memory model described in Sect. 4.1 is available in the Isabelle Archive of Formal Proofs [34]. The artefact of the evaluation provided in Sect. 5, which includes Gillian-CHERI-C itself, CHERI-ESBMC, and other tools, is archived in the Zenodo open-access repository [35].

References

1. CHERI C Tests. https://github.com/CTSRD-CHERI/cheri-c-tests
2. cheri-compressed-cap. https://github.com/CTSRD-CHERI/cheri-compressed-cap
3. CHERI RISC-V Sail model. https://github.com/CTSRD-CHERI/sail-cheri-riscv
4. CHERI: The Arm Morello Board, https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html
5. CHERI: The Digital Security by Design (DSbD) Initiative, https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/dsbd.html
6. Digital Security by Design Challenge – UKRI, https://www.ukri.org/our-work/our-main-funds/industrial-strategy-challenge-fund/artificial-intelligence-and-data-economy/digital-security-by-design-challenge/
7. fix the behaviour of free, https://github.com/GillianPlatform/Gillian/commit/6fa87b046f8d8f328c20b89cbdff1a00944da3fe, GillianPlatform/Gillian@ 6fa87b0
8. Morello Sail specification. https://github.com/CTSRD-CHERI/sail-morello
9. Sail model of CHERI-MIPS ISA. https://github.com/CTSRD-CHERI/sail-cheri-mips
10. SCorCH: Secure Code for Capability Hardware, https://scorch-project.github.io
11. Armv8.5-A Memory Tagging Extension. Tech. rep. (Jun 2021), https://documentation-service.arm.com/static/624ea580caabfd7b3c13e23f?token=
12. ARM Ltd.: Arm Architecture Reference Manual Supplement Morello for A-Profile Architecture (2022), https://documentation-service.arm.com/static/61e577e1b691546d37bd38a0?token=
13. Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019)
14. Beeren, J., Fernandez, M., Gao, X., Klein, G., Kolanski, R., Lim, J., Lewis, C., Matichuk, D., Sewell, T.: Finite Machine Word Library. Archive of Formal Proofs (Jun 2016), https://isa-afp.org/entries/Word_Lib.html, Formal proof development
15. Braußе, F., Shmarov, F., Menezes, R., Gadelha, M.R., Korovin, K., Reger, G., Cordeiro, L.C.: ESBMC-CHERI: Towards Verification of C Programs for CHERI Platforms with ESBMC. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 773–776. ISSTA 2022, Association for Computing Machinery, New York, NY, USA (2022)

16. Calcagno, C., O'Hearn, P.W., Yang, H.: Local Action and Abstract Separation Logic. In: 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007). pp. 366–378 (2007)
17. Chisnall, D.: Towards a Safe, High-Performance Heap Allocator (Sep 2022), https://soft-dev.org/events/cheritech22/slides/Chisnall.pdf, presented at CHERI Technical Workshop 2022
18. Chisnall, D., Rothwell, C., Watson, R.N., Woodruff, J., Vadera, M., Moore, S.W., Roe, M., Davis, B., Neumann, P.G.: Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. SIGPLAN Not. **50**(4), 117–130 (Mar 2015)
19. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A Precise Yet Efficient Memory Model For C. Electronic Notes in Theoretical Computer Science **254**, 85–103 (2009). https://doi.org/https://doi.org/10.1016/j.entcs.2009.09.061, proceedings of the 4th International Workshop on Systems Software Verification (SSV 2009)
20. Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.: Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 927–942. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020)
21. Haftmann, F.: Code generation from Isabelle/HOL theories (Dec 2021), https://isabelle.in.tum.de/doc/codegen.pdf
22. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data Refinement in Isabelle/HOL. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving. pp. 100–115. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_10
23. Klein, G., Kolanski, R., Boyton, A.: Mechanised Separation Algebra. In: Beringer, L., Felty, A. (eds.) Interactive Theorem Proving. pp. 332–337. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_22
24. Krebbers, R.: A Formal C Memory Model for Separation Logic. Journal of Automated Reasoning **57**(4), 319–387 (Dec 2016). https://doi.org/10.1007/s10817-016-9369-1
25. Krebbers, R., Leroy, X., Wiedijk, F.: Formal C Semantics: CompCert and the C Standard. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving. pp. 543–548. Springer International Publishing, Cham (2014)
26. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (Jun 2012)
27. Lochbihler, A.: Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving. pp. 116–132. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_11
28. Maksimovic, P., Ayoun, S.E., Santos, J.F., Gardner, P.: Gillian, part II: real-world verification for javascript and C. In: Silva, A., Leino, K.R.M. (eds.) Proceedings of the 33rd Computer Aided Verification International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Part II. Lecture Notes in Computer Science, vol. 12760, pp. 827–850. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_38
29. Maksimovic, P., Santos, J.F., Ayoun, S.E., Gardner, P.: Gillian: A Multi-Language Platform for Unified Symbolic Analysis (2021). https://doi.org/10.48550/ARXIV.2105.14769, https://arxiv.org/abs/2105.14769

30. Memarian, K., Gomes, V.B.F., Davis, B., Kell, S., Richardson, A., Watson, R.N.M., Sewell, P.: Exploring C Semantics and Pointer Provenance. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019)
31. Miller, M.: Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape (Feb 2019), https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf, presented at BlueHat IL
32. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. [ecture Notes in Computer Science, Springer (2002). https://doi.org/10.1007/3-540-45949-9
33. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371078, https://doi.org/10.1145/3371078
34. Park, S.H.: A Formal CHERI-C Memory Model. Archive of Formal Proofs (Nov 2022), https://isa-afp.org/entries/CHERI-C_Memory_Model.html, Formal proof development
35. Park, S.H., Pai, R., Melham, T.: Artifact for Paper A formal CHERI-C Semantics for Verification (Jan 2023). https://doi.org/10.5281/zenodo.7504675, https://doi.org/10.5281/zenodo.7504675
36. Richardson, A.: Porting C/C++ software to Morello (Sep 2022), https://soft-dev.org/events/cheritech22/slides/Richardson.pdf, presented at CHERI Technical Workshop 2022
37. Santos, J.F., Maksimovic, P., Ayoun, S.E., Gardner, P.: Gillian: Compositional Symbolic Execution for All. CoRR **abs/2001.05059** (2020), https://arxiv.org/abs/2001.05059
38. Tuch, H.: Formal Verification of C Systems Code. Journal of Automated Reasoning **42**(2), 125–187 (Apr 2009). https://doi.org/10.1007/s10817-009-9120-2
39. Watson, R., Laurie, B., Richardson, A.: Assessing the Viability of an Open- Source CHERI Desktop Software Ecosystem. Tech. rep., Capabilities Limited (Sep 2021), https://www.capabilitieslimited.co.uk/pdfs/20210917-capltd-cheri-desktop-report-version1-FINAL.pdf
40. Watson, R.N.M., Neumann, P.G., Woodruff, J., Roe, M., Almatary, H., Anderson, J., Baldwin, J., Barnes, G., Chisnall, D., Clarke, J., et al.: Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Tech. rep., University of Cambridge, Cambridge, England (Oct 2020), https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf
41. Watson, R.N.M., Richardson, A., Davis, B., Baldwin, J., Chisnall, D., Clarke, J., Filardo, N., Moore, S.M., Napierala, E., Sewell, P., Neumann, P.G.: CHERI C/C++ Programming Guide. Tech. rep., University of Cambridge, Cambridge, England (Jun 2020), https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf
42. Wesley Filardo, N., Gutstein, B.F., Woodruff, J., Ainsworth, S., Paul-Trifu, L., Davis, B., Xia, H., Tomasz Napierala, E., Richardson, A., Baldwin, J., Chisnall, D., Clarke, J., Gudka, K., Joannou, A., Theodore Markettos, A., Mazzinghi, A., Norton, R.M., Roe, M., Sewell, P., Son, S., Jones, T.M., Moore, S.W., Neumann, P.G., Watson, R.N.M.: Cornucopia: Temporal Safety for CHERI Heaps. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 608–625 (2020). https://doi.org/10.1109/SP40000.2020.00098

43. Woodruff, J., Joannou, A., Xia, H., Fox, A., Norton, R.M., Chisnall, D., Davis, B., Gudka, K., Filardo, N.W., Markettos, A.T., Roe, M., Neumann, P.G., Watson, R.N.M., Moore, S.W.: CHERI Concentrate: Practical Compressed Capabilities. IEEE Transactions on Computers **68**(10), 1455–1469 (2019). https://doi.org/10.1109/TC.2019.2914037
44. Woodruff, J., Watson, R.N.M., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R., Roe, M.: The CHERI Capability Model: Revisiting RISC in an Age of Risk. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). pp. 457–468. IEEE (Jun 2014)

Automated Verification for Real-Time Systems via Implicit Clocks and an Extended Antimirov Algorithm

Yahui Song(✉) and Wei-Ngan Chin

School of Computing, National University of Singapore, Singapore, Singapore
{yahuis,chinwn}@comp.nus.edu.sg

Abstract. The correctness of real-time systems depends both on the correct functionalities and the realtime constraints. To go beyond the existing Timed Automata based techniques, we propose a novel solution that integrates a modular Hoare-style forward verifier with a term rewriting system (TRS) on *Timed Effects* (*TimEffs*). The main purposes are to: increase the expressiveness, dynamically manipulate clocks, and efficiently solve clock constraints. We formally define a core language C^t, generalizing the real-time systems, modeled using mutable variables and timed behavioral patterns, such as *delay*, *timeout*, *interrupt*, *deadline*. Secondly, to capture real-time specifications, we introduce *TimEffs*, a new effects logic, that extends *regular expressions* with dependent values and arithmetic constraints. Thirdly, the forward verifier reasons temporal behaviors – expressed in *TimEffs* – of target C^t programs. Lastly, we present a purely algebraic TRS, i.e., an extended *Antimirov algorithm*, to efficiently check language inclusions between *TimEffs*. To demonstrate the feasibility of our proposal, we prototype the verification system; prove its soundness; report on case studies and experimental results.

1 Introduction

During the last three decades, a popular approach for specifying real-time systems has been based on Timed Automata (TAs) [1]. TAs are powerful in designing real-time models via explicit clocks, where real-time constraints are captured by explicitly setting/resetting clock variables. A number of automatic verification tools for TAs have proven to be successful [2,3,4,5]. Industrial case studies show that requirements for real-time systems are often structured into phases, which are then composed sequentially, in parallel, alternatively [6,7]. TAs lack high-level compositional patterns for hierarchical design; moreover, users often need to manipulate clock variables with carefully calculated clock constraints manually. The process is tedious and error-prone.

There have been some translation-based approaches on building verification support for compositional timed-process representations. For example, Timed Communicating Sequential Process (TCSP), Timed Communicating Object-Z (TCOZ) and *Statechart* based hierarchical Timed Automata are well suited for presenting compositional models of complex real-time systems. Prior works [8,9] systematically translate TCSP/TCOZ/Statechart models to flat TAs so that the

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 569–587, 2023.
https://doi.org/10.1007/978-3-031-30823-9-29

model checker Uppaal [3] can be applied. However, possible insufficiencies are: the expressiveness power is limited by the finite-state automata; and there is always a gap between the verified logic and the actual code implementation.

In this work, we investigate an alternative approach for verifying real-time systems. We propose a novel temporal specification language, Timed Effects (*TimEffs*), which enables a compositional verification via a Hoare-style forward verifier and a term rewriting system (TRS). More specifically, we specify system behaviors in the form of *TimEffs*, which integrates the Kleene Algebra with dependent values and arithmetic constraints, to provide real-time abstractions into traditional linear temporal logics. For example, one safety property, *"The event* `Done` *will be triggered no later than one time unit"*[1], is expressed in *TimEffs* as: $\Phi \triangleq$ `0≤t<1` $\wedge$ `(_* · Done)#t`. Here $\wedge$ connects the arithmetic formula and the timed trace; the operator `#` binds time variables to traces (here `t` is a time bound of `(_* · Done)`); `_` is a wildcard matching to any event; Kleene star $\star$ denotes a trace repetition. The above formula Φ corresponds to ` $_{[0,1)}$Done' in metric temporal logic (MTL), reads *"within one time unit,* `Done` *finally happens"*. Furthermore, the time bounds can be dependent on the program inputs, as shown in Fig. 1.

```
1  void addOneSugar()
2  /* req: true ∧ _*
3    ens: t>1 ∧ ϵ # t */
4    timeout (() , 1); }
5
6  void addNSugar (int n)
7  /* req: true ∧ _*
8    ens: t≥n ∧ EndSugar # t */
9    if (n == 0)
10     event ["EndSugar"];}
11   else
12     addOneSugar();
13     addNSugar (n-1);}}
```

Fig. 1. Value-dependent specification.

Function `addNSugar` takes a parameter `n`, representing the portion of the sugar to add. When `n=0`, it raises an event `EndSugar` to mark the end of the process. Otherwise, it adds one portion of the sugar by calling `addOneSugar()`, then recursively calls `addNSugar` with parameter `n-1`. The use of `timeout(e, d)` is standard [11], which executes a block of code `e` after the specified time `d`. Therefore, the time spent on adding one portion of the sugar is more than one time unit. Note that ϵ`#t` refers to an empty trace which takes time `t`. Both preconditions require no arithmetic constraints and no temporal constraints upon the history traces. The postcondition of `addNSugar(n)` indicates that the method generates a finite trace where `EndSugar` takes a no less than `n` time-units delay to finish.

Although these examples are simple, they show the benefits of deploying value-dependent time bounds, which is beyond the capability of TAs. Essentially, *TimEffs* define symbolic TAs, which stands for a set (possibly infinite) of concrete transition systems. Moreover, we deploy a Hoare-style forward verifier to soundly reason about the behaviors from the source level, with respect to the well-defined operational semantics. This approach provides a *direct* (opposite to the techniques which require manual and remote modeling processes), and modular verification – where modules can be replaced by their already verified properties – for real-time systems, which are not possible by any existing tech-

[1] In this paper, we pretend time is discrete and only integral values. However, it's just as easy to represent continuous time by letting time variables assume real values [10].

niques. Furthermore, we develop a novel TRS, which is inspired by Antimirov and Mosses' algorithm[2] [12] but solving the language inclusions between more expressive *TimEffs*. In short, the main contributions of this work are:

1. **Language Abstraction:** we formally define a core language C^t, by defining its syntax and operational semantics, generalizing the real-time systems with mutable variables and timed behavioral patterns, e.g., *delay*, *timeout*, *deadline*.
2. **Novel Specification:** we propose *TimEffs*, by defining its syntax and semantics, gaining the expressive power beyond traditional linear temporal logics.
3. **Forward Verifier:** we establish a sound effect system to reason about temporal behaviors of given programs. The verifier triggers the back-end solver TRS.
4. **Efficient TRS:** we present the rewriting rules to (dis)prove the inclusion relations between the actual behaviors and the given specifications, both in *TimEffs*.
5. **Implementation and Evaluation:** we prototype the automated verification system, prove its soundness, report on case studies and experimental results.

2 Overview

An overview of our automated verification system is given in Fig. 2. The system consists of a forward verifier and a TRS, i.e., the rounded boxes. The input of the forward verifier is a C^t program annotated with temporal specifications written in *TimEffs*. The input of the TRS is a pair of effects LHS and RHS, referring to the inclusion LHS $\sqsubseteq$ RHS[3] to be checked *(LHS and RHS refer to left/right-hand-side effects respectively)*. The forward verifier calls TRS to solve proof obligations. Next, we use Fig. 3 to highlight our main methodologies, which simulates a coffee machine, that dynamically adds sugar based on the user's input number.

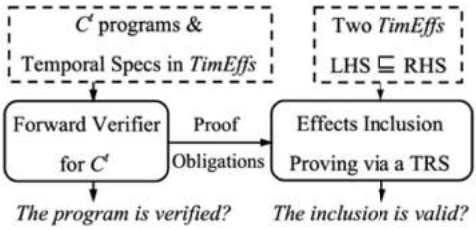


Fig. 2. System Overview.

2.1 ***TimEffs.*** We define Hoare-triple style specifications (enclosed in `/*...*/`) for each function, which leads to a compositional verification strategy, where static checking can be done locally. The precondition of `makeCoffee` specifies that the input value `n` is non-negative, and it *requires* that before entering into this function, this history trace must contain the event `CupReady` on the tail. The verification fails if the precondition is not satisfied at the caller sites. Line 17 sets a five time-units deadline (i.e., maximum 5 portion of sugar per coffee) while calling `addNSugar` (defined in Fig. 1); then emits event `Coffee` with a deadline, indicating the pouring coffer process takes no more than four time-units. The precondition of `main` requires no arithmetic constraints (expressed as `true`) and an empty history trace. The postcondition of `main` specifies that before the final

[2] Antimirov and Mosses' algorithm was designed for deciding the inequalities of regular expressions based on an axiomatic algorithm of the algebra of regular sets.
[3] The *TimEffs* inclusion relation $\sqsubseteq$ is formally defined in Definition 3.

Done happens, there is no occurrence of Done (! indicates the absence of events); and the whole process takes no more than nine time-units to hit the final event.

```
14 void makeCoffee (int n)
15 /* req: n≥0 ∧ _*· CupReady
16   ens: n≤t≤5 ∧ t'≤4 ∧
         (EndSugar # t) · (Coffee # t') */
17   deadline (addNSugar(n), 5);
18   deadline (event["Coffee"],4);}
19
20 int main ()
21 /* req: true ∧ ϵ
22   ens: t≤9 ∧ ((!Done)* # t) · Done */
23   event["CupReady"];
24   makeCoffee (3);
25   event["Done"];}
```

Fig. 3. To make coffee with three portions of sugar within nine time units.

TimEffs support more features such as *disjunctions*, *guards*, *parallelism* and *assertions*, etc (cf. Sec. 3.3), providing detailed information upon: branching properties: different arithmetic conditions on the inputs lead to different effects; and required history traces: by defining the prior effects in precondition. These capabilities are beyond traditional timed verification, and cannot be fully captured by any prior works [8,9,2,3,4,5]. Nevertheless, the increase in expressive power needs support from finer-grind reasoning and a more sophisticated back-end solver, discharged by our forward verifier and TRS.

1. void addOneSugar() // *initialize the state using the function precondition.*
   $\Phi_C{=}\Phi_{re}^{addOneSugar(n)} = \{\texttt{true} \wedge \_^*\}$ [*FV-Meth*]
2. timeout ((), 1);}
   $\Phi'_C{=}\{\texttt{t1>1} \wedge \_^* \cdot (\epsilon \texttt{ \# t1})\}$ [*FV-Timeout*]
3. $\Phi'_C \sqsubseteq \Phi_{pre}^{addOneSugar(n)} \cdot \Phi_{post}^{addOneSugar(n)} \Leftrightarrow \texttt{t1>1} \wedge \_^* \cdot (\epsilon\texttt{\#t1}) \sqsubseteq \texttt{t>1} \wedge \_^* \cdot (\epsilon\texttt{\#t})$

---

4. void addNSugar (int n) // *initialize the state using the function precondition.*
   $\Phi_C{=}\Phi_{re}^{addNSugar(n)} = \{\texttt{true} \wedge \_^*\}$ [*FV-Meth*]
5. if (n == 0)
   $\{\texttt{n=0} \wedge \_^*\}$ [*FV-Cond*]
6. event ["EndSugar"];}
   $\{\texttt{n=0} \wedge \_^* \cdot \texttt{EndSugar}\}$ [*FV-Event*]
7. else
   $\{\texttt{n≠0} \wedge \_^*\}$ [*FV-Cond*]
8. addOneSugar();
   $\{\texttt{n≠0∧t2>1} \wedge \_^* \cdot (\epsilon \texttt{ \# t2})\}$ [*FV-Call*]
9. addNSugar (n-1);}
   $\texttt{n≠0∧t2>1} \wedge \_^* \cdot (\epsilon \texttt{ \# t}) \sqsubseteq \Phi_{re}^{addNSugar(n-1)}$ // *TRS: precondition checked.*
   $\{\texttt{n≠0∧t2>1} \wedge \_^* \cdot (\epsilon \texttt{ \# t2}) \cdot \Phi_{ost}^{addNSugar(n-1)}\}$ [*FV-Call*]
10. $\Phi'_C{=}\ (\texttt{n=0} \wedge \_^*\cdot\texttt{Sugar}) \vee (\texttt{n≠0∧t2>1} \wedge \_^*\cdot(\epsilon\texttt{\#t2})\cdot\Phi_{ost}^{addNSugar(n-1)})$ [*FV-Cond*]
11. $\Phi'_C \sqsubseteq \Phi_{pre}^{addNSugar(n)} \cdot \Phi_{post}^{addNSugar(n)} \Leftrightarrow$ //*TRS: postcondition checked, cf. Table 1*
    $(\texttt{n=0} \wedge \texttt{Sugar}) \vee (\texttt{n≠0∧t2>1} \wedge (\epsilon \texttt{ \# t2}) \cdot \Phi_{ost}^{addNSugar(n-1)}) \sqsubseteq \Phi_{ost}^{addNSugar(n)}$

Fig. 4. The forward verification examples (t1 and t2 are fresh time variables).

2.2 Forward Verification. Fig. 4 demonstrates the forward verification of functions addOneSugar and addNSugar, defined in Fig. 1. The effects states are captured in the form of $\{\Phi_C\}$. To facilitate the illustration, we label the steps

by (1) to (11), and mark the deployed forward rules (cf. Sec. 4.1) in [gray]. The initial states (1) and (4) are obtained from the preconditions, by the [*FV-Meth*] rule. States (5)(7)(10) are obtained by [*FV-Cond*], which enforces the conditional constraints into the effects states, and unions the effects accumulated from two branches. State (6) is obtained by [*FV-Event*], which concatenates an event to the current effects. The intermediate states (8) and (9) are obtained by [*FV-Call*]. Before each function call, [*FV-Call*] invokes the TRS to check whether the current effects states satisfy callees' preconditions. If it is not satisfied, the verification fails; otherwise, it concatenates the callee's postcondition to the current states (the precondition check for step (8) is omitted here).

State (2) is obtained by [*FV-Timeout*], which adds a lower time-bound to an empty trace. After these state transformations, steps (3) and (11) invoke the TRS to check the inclusions between the final effects and the declared postconditions.

2.3 The TRS. Having *TimEffs* to be the specification language, and the forward verifier to reason about the actual behaviors, we are interested in the following verification problem: Given a program $\mathcal{P}$, and a temporal specification Φ', does the inclusions $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ holds? Typically, checking the inclusion/entailment between the concrete program effects $\Phi^{\mathcal{P}}$ and the expected property Φ' proves that: the program $\mathcal{P}$ will never lead to unsafe traces which violate Φ'.

Our TRS is an extension of Antimirov and Mosses's algorithm [12], which can be deployed to decide inclusions of two regular expressions (REs) through an iterated process of checking inclusions of their *partial derivatives* [13]. There are two basic rules: [*Disprove*] infers false from trivially inconsistent inclusions; and [*Unfold*] applies Definition 2 to generate new inclusions.

Definition 1 (Derivative). *Given any formal language S over an alphabet Σ and any string $u \in \Sigma^*$, the derivative of S with respect to u is defined as:*

$$u^{-1}S = \{w \in \Sigma^* \mid uw \in S\}.$$

Definition 2 (REs Inclusion). *For REs r and s, $r \preceq s \Leftrightarrow \forall(\mathtt{A} \in \Sigma).\mathtt{A}^{-1}(r) \preceq \mathtt{A}^{-1}(s)$.*

Definition 3 (*TimEffs* Inclusion). *For* TimEffs Φ_1 *and* Φ_2,

$$\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow \forall \mathtt{A}.\forall \mathtt{t} \geq 0.\ (\mathtt{A\#t})^{-1}\Phi_1 \sqsubseteq (\mathtt{A\#t})^{-1}\Phi_2.$$

Similarly, we defined Definition 3 for unfolding the inclusions between *TimEffs*, where $(\mathtt{A\#t})^{-1}\Phi$ is the partial derivative of Φ w.r.t the event `A` with the time bound `t`. Termination of the rewriting is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* (cf. Table 5) [14]. Next, we use Table 1 to demonstrate how the TRS automatically proves the final effects of `main` satisfying its postcondition (shown at step (11) in Fig. 4). We mark the rewriting rules (cf. Sec. 5) in [gray].

In Table 1, step ① renames the time variables to avoid the name clashes between the antecedent and the consequent. Step ② splits the proof tree into two branches, according to the different arithmetic constraints, by rule [LHS-OR]. In the first branch, step ③ eliminates the event `ES` from the head of both sides, by rule [UNFOLD]. Step ④ proves the inclusion, because evidently the consequent $\mathtt{tR} \geq 0 \wedge \epsilon \mathtt{\#tR}$ contains ϵ when `tR=0`. In the second branch, step ⑤ eliminates a

Table 1. An inclusion proving example. (I) is the right hand side sub-tree of the the main rewriting proof tree. (ES stands for the event EndSugar)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{n=0} \wedge \epsilon \sqsubseteq \texttt{tR}\geq\texttt{0} \wedge \epsilon \texttt{ \# tR}}\ ④\ \text{[PROVE]}}{\texttt{n=0} \wedge \cancel{\texttt{ES}} \sqsubseteq \texttt{tR}\geq\texttt{0} \wedge \texttt{ES\#tR}}\ ③\ \text{[UNFOLD]} \quad (I)}{(\texttt{n=0}\wedge\texttt{ES}) \vee (\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)}\wedge \epsilon\texttt{\#t2} \cdot \texttt{ES\#tL}) \sqsubseteq \texttt{tR}\geq\texttt{n} \wedge \texttt{ES\#tR}}\ ②\ \text{[LHS-OR]}}{(\texttt{n=0} \wedge \texttt{ES}) \vee (\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1} \wedge (\epsilon \texttt{ \# t2}) \cdot \Phi_{post}^{addNSugar(n-1)}) \sqsubseteq \Phi_{post}^{addNSugar(n)}}\ ①\ \text{[RENAME]}$$

$$\dfrac{\dfrac{\dfrac{}{\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)} \wedge \texttt{tL=(tR-t2)} \Rightarrow \texttt{tR}\geq\texttt{n}}\ ⑦\ \text{[PROVE]}}{\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)} \wedge \epsilon \sqsubseteq \texttt{tR}\geq\texttt{n} \wedge \epsilon}\ ⑥\ \text{[UNFOLD]}\ \pi_u\texttt{:tL=(tR-t2)}}{\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)} \wedge \cancel{\texttt{ES\#tL}} \sqsubseteq \texttt{tR}\geq\texttt{n} \wedge \cancel{\texttt{ES\#(tR-t2)}}}\ ⑤\ \text{[UNFOLD]}$$
$$\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)} \wedge \cancel{\epsilon\texttt{\#t2}} \cdot \texttt{ES\#tL} \sqsubseteq \texttt{tR}\geq\texttt{n} \wedge \texttt{ES\#tR}$$

(I)

time duration ϵ#t2 from both sides. Therefore the rule [UNFOLD] subtracts a time duration from the consequent, i.e., (tR-t2). Similarly, step ⑥ eliminates ES#tL from the both sides, adding tL=(tR-t2) to the unification constraints. Step ⑦ proves t2>1∧tL≥(n-1)∧tL=(tR-t2)⇒tR≥n [4]; therefore, the proof succeed.

2.4 Verifying the Fischer's Mutual Exclusion Protocol. Fig. 5 presents the classical Fischer's mutually exclusion protocol, in C^t. Global variables x and cs indicate *'which process attempted to access the critical section most recently'* and *'the number of processes accessing the critical section'* respectively. The main procedure is a parallel composition of three processes, where d and e are two constants. Each process attempts to enter the critical section when x is -1, i.e. no other process is currently attempting. Once the process is active (i.e., reaches line 6), it sets x to its identity number i within d time units, captured by deadline(...,d). Then it idles for e time units, captured by delay(e) and then checks whether x still equals to i. If so, it safely enters the critical section. Otherwise, it restarts from the beginning. Quantitative timing constraint d<e plays an important role in this algorithm to guarantee mutual exclusion. One way to prove mutual exclusion is to show that cs≤1 is always true. Or, using event temporal logic, we can show that the occurrence of Critical always indicates the next event is Exit. We show in Sec. 6 that our prototype system can verify such algorithms symbolically.

```
1 var x := -1;
2 var cs:= 0;
3
4 void proc (int i) {
5  [x=-1] //block waiting until true
6  deadline(event["Update"(i)]{x:=i},d);
7    delay (e);
8    if (x==i) {
9      event["Critical"(i)]{cs:=cs+1};
10     event["Exit"(i)]{cs:=cs-1;x:=-1};
11     proc (i);
12   } else {proc (i);}}
13
14 void main ()
15 /* req: d<e ∧ ϵ          ens_c: true∧(cs≤1)*
16   ens_b: true∧ ((_*).Critical.Exit.(_*))* */
17 { proc(0) || proc(1) || proc(2); }
```

Fig. 5. Fischer's mutually exclusion algorithm.

[4] The proof obligations for arithmetic constraints are discharged by the Z3 solver [15].

3 Language and Specifications

3.1 The Target Language

We define the core language C^t in Fig. 6, which is built based on C syntax and provides support for timed behavioral patterns.

$$
\begin{array}{lr}
(\textit{Program}) & \mathcal{P} ::= (\alpha^{\cup}, \textit{meth}^{\cup}) \\
(\textit{Types}) & \iota ::= \textit{int} \mid \textit{bool} \mid \textit{void} \\
(\textit{Method}) & \textit{meth} ::= \iota\ mn\ (\iota\ x)^{\cup}\ \{\textbf{req}\ \Phi_{pre}\ \textbf{ens}\ \Phi_{post}\}\ \{e\} \\
(\textit{Values}) & v ::= () \mid c \mid b \mid x \\
(\textit{Assignment}) & \alpha ::= x := v \\
(\textit{Expressions}) & e ::= v \mid \alpha \mid [v]e \mid mn(v^{\cup}) \mid e_1; e_2 \mid e_1 || e_2 \mid \textit{if}\ v\ e_1\ e_2 \mid \texttt{event}[\texttt{A}(v, \alpha^{\cup})] \\
 & \quad \mid \texttt{delay}[v] \mid e_1\ \texttt{timeout}[v]\ e_2 \mid e\ \texttt{deadline}[v] \mid e_1\ \texttt{interrupt}[v]\ e_2 \\
(\textit{Terms}) & t ::= c \mid x \mid t_1 \texttt{+} t_2 \mid t_1 \texttt{-} t_2
\end{array}
$$

$c \in \mathbb{Z}$ $\quad b \in \mathbb{B}$ $\quad mn, x \in \textbf{var}$ $\quad (\textit{Action labels})\ \texttt{A} \in \Sigma$

Fig. 6. A core first-order imperative language with timed constructs via implicit clocks.

Here, c and b stand for integer and Boolean constants, mn and x are meta-variables, drawn from **var** (the countably infinite set of arbitrary distinct identifiers). A program $\mathcal{P}$ comprises a list of global variable initializations α^* and a list of method declarations $\textit{meth}^*$. Here, we use the $*$ superscript to denote a finite list of items, for example, x^* refers to a list of variables, $x_1, ..., x_n$. Each method $\textit{meth}$ has a name mn, an expression-oriented body e, also is associated with a precondition Φ_{pre} and a postcondition Φ_{post} (specification syntax is given in Fig. 7). C^t allows each iterative loop to be optimized to an equivalent tail-recursive method, where mutation on parameters is made visible to the caller.

Expressions comprise: values v; guarded processes $[v]e$, where if v is true, it behaves as e, else it idles until v becomes true; method calls $mn(v^*)$; sequential composition $e_1; e_2$; parallel composition $e_1 || e_2$, where e_1 and e_2 may communicate via shared variables; conditionals $\textit{if}\ \ v\ \ e_1\ \ e_2$; and event raising expressions $\texttt{event}[\texttt{A}(v, \alpha^*)]$ where the event $\texttt{A}$ comes from the finite set of event labels Σ. Without loss of generality, events can be further parametrized with one value v and a set of assignments α^* to update the mutable variables. Moreover, a number of timed constructs can be used to capture common real-time system behaviors, which are explained via operational semantics rules in Sec. 3.2.

3.2 Operational Semantics of C^t

To build the semantics of the system model, we define the notion of a configuration in Definition 4, to capture the global system state during system execution.

Definition 4 (System configuration). *A system configuration ζ is a pair $(\mathcal{S}, e)$ where $\mathcal{S}$ is a variable valuation function (or a stack) and e is an expression.*

A transition of the system is of the form $\zeta \xrightarrow{l} \zeta'$ where ζ and ζ' are the system configurations before and after the transition respectively. Transition labels l include: d, denoting a non-negative integer; τ, denoting an invisible event; $\texttt{A}$,

denoting an observable event. For example, $\zeta \xrightarrow{\mathrm{d}} \zeta'$ denotes a d time-units elapse. Next, we present the firing rules, associated with timed constructs.

Process `delay`$[v]$ idles for exactly t time units. Rule $[delay_1]$ states that the process may idle for any amount of time given it is less than or equal to t; Rule $[delay_2]$ states that the process terminates immediately when t becomes 0.

$$\frac{\mathrm{d}{\leq}v}{(\mathcal{S}, \texttt{delay}[v]) \xrightarrow{\mathrm{d}} (\mathcal{S}, \texttt{delay}[v\text{-d}])}\ [delay_1] \qquad \frac{}{(\mathcal{S}, \texttt{delay}[0]) \xrightarrow{\tau} (\mathcal{S}, ())}\ [delay_2]$$

In e_1 `timeout`$[v]$ e_2, the first observable event of e_1 shall occur before t time units; otherwise, e_2 takes over the control after exactly t time units. Note that the usage of timeout in Fig. 1 is a special case where e_1 never starts by default.

$$\frac{(\mathcal{S}, e_1) \xrightarrow{\mathtt{A}} (\mathcal{S}', e_1')}{(\mathcal{S}, e_1\ \texttt{timeout}[v]\ e_2) \xrightarrow{\mathtt{A}} (\mathcal{S}', e_1')}\ [to_1] \qquad \frac{(\mathcal{S}, e_1) \xrightarrow{\tau} (\mathcal{S}', e_1')}{(\mathcal{S}, e_1\ \texttt{timeout}[v]\ e_2) \xrightarrow{\tau} (\mathcal{S}', e_1'\ \texttt{timeout}[v]e_2)}\ [to_2]$$

$$\frac{(\mathcal{S}, e_1) \xrightarrow{\mathrm{d}} (\mathcal{S}, e_1') \qquad (\mathrm{d}{\leq}v)}{(\mathcal{S}, e_1\ \texttt{timeout}[v]\ e_2) \xrightarrow{\mathrm{d}} (\mathcal{S}, e_1'\ \texttt{timeout}[v\text{-d}]e_2)}\ [to_3] \qquad \frac{}{(\mathcal{S}, e_1\ \texttt{timeout}[0]e_2) \xrightarrow{\tau} (\mathcal{S}, e_2)}\ [to_4]$$

Process `deadline` $[v]$ e behaves exactly as e except that it must terminate before t time units. The guarded process $[v]e$ behaves as e when v is true, otherwise it idles until v becomes true. Process e_1 `interrupt`$[v]$ e_2 behaves as e_1 until t time units, and then e_2 takes over. We leave the rest rules in [16].

$$\frac{(\mathcal{S}, e) \xrightarrow{\mathtt{A}/\tau} (\mathcal{S}', e')}{(\mathcal{S}, \texttt{deadline}[v]\ e) \xrightarrow{\mathtt{A}/\tau} (\mathcal{S}', \texttt{deadline}[v]\ e')}\ [ddl_1] \qquad \frac{(\mathcal{S}, e) \xrightarrow{l} (\mathcal{S}', v)}{(\mathcal{S}, \texttt{deadline}[v]\ e) \xrightarrow{l} (\mathcal{S}', v)}\ [ddl_2]$$

$$\frac{\mathcal{S} \models (v{=}true)}{(\mathcal{S}, [v]e) \xrightarrow{\tau} (\mathcal{S}, e)}\ [gu_1] \qquad \frac{(\mathcal{S}, e) \xrightarrow{\mathrm{d}} (\mathcal{S}, e') \quad (\mathrm{d}{\leq}v)}{(\mathcal{S}, \texttt{deadline}[v]\ e) \xrightarrow{\mathrm{d}} (\mathcal{S}, \texttt{deadline}[v\text{-d}]\ e')}\ [ddl_3]$$

$$\frac{\mathcal{S} \not\models (v{=}true)}{(\mathcal{S}, [v]e) \xrightarrow{\tau} (\mathcal{S}, [v]e)}\ [gu_2] \qquad \frac{(\mathcal{S}, e_1) \xrightarrow{\mathtt{A}/\tau} (\mathcal{S}', e_1')}{(\mathcal{S}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{\mathtt{A}/\tau} (\mathcal{S}', e_1'\ \texttt{interrupt}[v]\ e_2)}\ [int_1]$$

$$\frac{(\mathcal{S}, e_1) \xrightarrow{l} (\mathcal{S}', v)}{(\mathcal{S}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{l} (\mathcal{S}', v)}\ [int_2] \qquad \frac{}{(\mathcal{S}, e_1\ \texttt{interrupt}[0]\ e_2) \xrightarrow{\tau} (\mathcal{S}, e_2)}\ [int_3]$$

$$\frac{(\mathcal{S}, e_1) \xrightarrow{\mathrm{d}} (\mathcal{S}, e_1') \quad (\mathrm{d}{\leq}v)}{(\mathcal{S}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{\mathrm{d}} (\mathcal{S}, e_1'\ \texttt{interrupt}[v\text{-d}]\ e_2)}\ [int_4]$$

3.3 The Specification Language

We plant *TimEffs* specifications into the Hoare-style verification system, using Φ_{pre} and Φ_{post} to capture the temporal pre/post conditions. As shown in Fig. 7, *TimEffs* can be constructed by a conditioned event sequence $\pi \wedge \theta$; or an effects disjunction $\Phi_1 \vee \Phi_2$. Timed sequences comprise *nil* ($\perp$); empty trace ϵ; single event ev; concatenation $\theta_1 \cdot \theta_2$; disjunction $\theta_1 \vee \theta_2$; parallel composition $\theta_1 || \theta_2$; a block waiting for a certain constraint to be satisfied $\pi?\theta$. We introduce a new operator `#`, and $\theta\#t$ represents the trace θ takes t time units to complete, where t

$$
\begin{array}{rl}
(\textit{Timed Effects}) & \Phi ::= \pi \wedge \theta \mid \Phi_1 \vee \Phi_2 \\
(\textit{Event Sequences}) & \theta ::= \bot \mid \epsilon \mid ev \mid \theta_1 \cdot \theta_2 \mid \theta_1 \vee \theta_2 \mid \theta_1 || \theta_2 \mid \pi ? \theta \mid \theta \# t \mid \theta^\star \\
(\textit{Events}) & ev ::= \mathtt{A}(v, \alpha^{\cup}) \mid \tau(\pi) \mid \overline{\mathtt{A}} \mid \_ \\
(\textit{Pure}) & \pi ::= \mathit{True} \mid \mathit{False} \mid \mathit{bop}(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg \pi \mid \pi_1 \Rightarrow \pi_2 \\
(\textit{Real-Time Terms}) & t ::= c \mid x \mid t_1 + t_2 \mid t_1 - t_2
\end{array}
$$

$c \in \mathbb{Z}$ $\quad x \in \mathbf{var}$ $\quad (\textit{Real Time Bound})\ \#$ $\quad (\textit{Kleene Star})\ \star$

Fig. 7. Syntax of *TimEffs*.

is a *real-time term*. A timed sequence also can be constructed by $\theta^\star$, representing zero or more times repetition of the trace θ. For single events, $\mathtt{A}(v, \alpha^*)$ stands for an observable event with label $\mathtt{A}$, parameterized by v, and the assignment operations α^*; $\tau(\pi)$ is an invisible event, parameterized with a pure formula π[5].

Events can also be $\overline{\mathtt{A}}$, referring to all events which are not labeled using $\mathtt{A}$; and a wildcard _, which matches to all the events. We use π to denote a pure formula which captures the (Presburger) arithmetic conditions on terms or program parameters. We use $\mathit{bop}(t_1, t_2)$ to represent binary atomic formulas of terms (including $=$, $>$, $<$, $\geq$ and $\leq$). Terms consist of constant integer values c; integer variables x; simple computations of terms, $t_1 + t_2$ and $t_1 - t_2$.

3.4 Semantic Model of Timed Effects

Let $d, \mathcal{S}, \varphi \models \Phi$ denote the model relation, i.e., a stack $\mathcal{S}$, a concrete execution trace φ take d time units to complete, and they satisfy the specification Φ.

$$
\begin{array}{ll}
d, \mathcal{S}, \varphi \models \Phi_1 \vee \Phi_2 & \textit{iff } d, \mathcal{S}, \varphi \models \Phi_1 \textit{ or } d, \mathcal{S}, \varphi \models \Phi_2 \\
d, \mathcal{S}, \varphi \models \pi \wedge \epsilon & \textit{iff } d{=}0 \textit{ and } [\![\pi]\!]_s{=}\mathit{True} \textit{ and } \varphi{=}[] \\
d, \mathcal{S}, \varphi \models \pi \wedge ev & \textit{iff } d{=}0 \textit{ and } [\![\pi]\!]_s{=}\mathit{True} \textit{ and } \varphi{=}[ev] \\
d, \mathcal{S}, \varphi \models \pi \wedge (\theta_1 \cdot \theta_2) & \textit{iff } \exists \varphi_1, \varphi_2.\ \varphi_1 {+\!+} \varphi_2{=}\varphi \textit{ and } \exists d_1, d_2.\ d_1{+}d_2{=}d \\
 & \quad \textit{s.t. } d_1, \mathcal{S}, \varphi_1 \models \pi \wedge \theta_1 \textit{ and } d_2, \mathcal{S}, \varphi_2 \models \pi \wedge \theta_2 \\
d, \mathcal{S}, \varphi \models \pi \wedge (\theta_1 \vee \theta_2) & \textit{iff } d, \mathcal{S}, \varphi \models \pi \wedge \theta_1 \textit{ or } d, \mathcal{S}, \varphi \models \pi \wedge \theta_2 \\
d, \mathcal{S}, \varphi \models \pi \wedge (ev_1 \cdot \theta_1) || (ev_2 \cdot \theta_2) & \textit{iff } d, \mathcal{S}, \varphi \models \pi \wedge ev_1 \cdot (\theta_1 || (ev_2 \cdot \theta_2)) \textit{ or} \\
 & \quad d, \mathcal{S}, \varphi \models \pi \wedge ev_2 \cdot ((ev_1 \cdot \theta_1) || \theta_2) \\
d, \mathcal{S}, \varphi \models \pi \wedge (ev \cdot \theta_1) || (ev \cdot \theta_2) & \textit{iff } d, \mathcal{S}, \varphi \models \pi \wedge ev \cdot (\theta_1 || \theta_2) \\
d, \mathcal{S}, \varphi \models \pi \wedge (\epsilon \# t_1) || (\epsilon \# t_2) & \textit{iff } d, \mathcal{S}, \varphi \models (\pi \wedge t_1 \geq t_2) \wedge (\epsilon \# t_1) \cdot (\epsilon \# (t_1 - t_2)) \textit{ or} \\
 & \quad d, \mathcal{S}, \varphi \models (\pi \wedge t_1 < t_2) \wedge (\epsilon \# t_2) \cdot (\epsilon \# (t_2 - t_1)) \\
d, \mathcal{S}, \varphi \models \pi \wedge \pi_1 ? \theta & \textit{iff } [\![\pi_1]\!]_s{=}\mathit{True}, d, \mathcal{S}, \varphi \models \pi \wedge \theta \textit{ or} \\
 & \quad [\![\pi_1]\!]_s{=}\mathit{False}, d, \mathcal{S}, \varphi \models \pi \wedge \pi_1 ? \theta \\
d, \mathcal{S}, \varphi \models \pi \wedge \theta \# t & \textit{iff } [\![\pi \wedge t \geq 0]\!]_s{=}\mathit{True},\ \exists \theta_1, \theta_2.\ \theta_1 \cdot \theta_2{=}\theta,\ \textit{fresh } t_1, t_2, \textit{s.t.} \\
 & \quad d, \mathcal{S}, \varphi \models (\pi \wedge t_1 \geq 0 \wedge t_2 \geq 0 \wedge t_1 + t_2 = t) \wedge (\theta_1 \# t_1) \cdot (\theta_2 \# t_2) \\
d, \mathcal{S}, \varphi \models \pi \wedge \theta^\star & \textit{iff } d, \mathcal{S}, \varphi \models \pi \wedge \epsilon \textit{ or } d, \mathcal{S}, \varphi \models \pi \wedge \theta \cdot \theta^\star \\
d, \mathcal{S}, \varphi \models \mathit{false} & \textit{iff } [\![\pi]\!]_s{=}\mathit{False} \textit{ or } \varphi{=}\bot
\end{array}
$$

Fig. 8. Semantics of *TimEffs*.

[5] The difference between $\tau(\pi)$ and $\pi?$ is: $\tau(\pi)$ marks an assertion which leads to false ($\bot$) if π is not satisfied, whereas $\pi?$ waits until π is satisfied.

To define the model, *var* is the set of program variables, *val* is the set of primitive values; and $d, \mathcal{S}, \varphi$ are drawn from the following concrete domains: d: $\mathbb{N}$, $\mathcal{S}$: *var*→*val* and φ: *list of event*. As shown in Fig. 8, ++ appends event sequences; [] describes the empty sequences, $[ev]$ represents the singleton sequence contains event ev; $[\![\pi]\!]_{\mathcal{S}}$=*True* represents π holds on the stack $\mathcal{S}$. Notice that, simple events, i.e., without #, are taken to be happening in instant time.

3.5 Expressiveness. *TimEffs* draw similarities to metric temporal logic (MTL), which is derived from LTL, where a set of non-negative real numbers is added to temporal modal operators. As shown in Table 2, we are able to encode MTL operators into *TimEffs*, making it more intuitive and readable. The basic modal operators are: $\triangleq$ for "globally"; $\overleftarrow{\triangleq}$ for "finally"; $\bigcirc$ for "next"; $\mathcal{U}$ for "until", and their past time reversed versions: $\overleftarrow{\triangleq}$; $\overleftarrow{\ }$; and $\ominus$ for "previous"; $\mathcal{S}$ for "since". I in MTL is the time interval with concrete upper/lower bounds; whereas in *TimEffs* they can be symbolic bounds which are dependent on program inputs.

Table 2. Examples for converting MTL formulae into *TimEffs* with $\mathtt{t}{\in}I$ applied.

Φ_{post}	$\triangleq_I \mathtt{A} \equiv (\mathtt{A}^\star)\mathtt{\#t}$	$\ _I\mathtt{A} \equiv (\_^\star \cdot \mathtt{A})\mathtt{\#t}$	$\bigcirc_I \mathtt{A} \equiv (\_)\mathtt{\#t} \cdot \mathtt{A}$	$\mathtt{A}\mathcal{U}_I\mathtt{B} \equiv (\mathtt{A}^\star)\mathtt{\#t} \cdot \mathtt{B}$
Φ_{pre}	$\overleftarrow{\triangleq}_I \mathtt{A} \equiv (\mathtt{A}^\star)\mathtt{\#t}$	$\overleftarrow{\ }_I\mathtt{A} \equiv (\mathtt{A} \cdot \_^\star)\mathtt{\#t}$	$\ominus_I \mathtt{A} \equiv \mathtt{A} \cdot ((\_)\mathtt{\#t})$	$\mathtt{A}\mathcal{S}_I\mathtt{B} \equiv \mathtt{B} \cdot ((\mathtt{A}^\star)\mathtt{\#t})$

4 Automated Forward Verification

4.1 Forward Rules

Forward rules are in the Hoare-style triples $\mathcal{S} \vdash \{\Pi, \Theta\}\ e\ \{\Pi', \Theta'\}$, where $\mathcal{S}$ is the stack environment; $\{\Pi, \Theta\}$ and $\{\Pi', \Theta'\}$ are program states, i.e., disjunctions of conditioned event sequence $\pi \wedge \theta$. The meaning of the transition is: $\{\Pi', \Theta'\} = \bigcup_{i=0}^{|\{\Pi,\Theta\}|-1} \{\Pi_i', \Theta_i'\}$ *where* $(\pi_i \wedge \theta_i) \in \{\Pi, \Theta\}$ *and* $\vdash \{\pi_i, \theta_i\}\ e\ \{\Pi_i', \Theta_i'\}$[6].

We here present the rules for time-related constructs and leave the rest rules in [16]. Rule [*FV-Delay*] creates a trace $\epsilon\mathtt{\#}t$, where t is fresh, and concatenates it to the current program state, together with the additional constraint $t{=}v$. Rule [*FV-Deadline*] computes the effects from e and adds an upper time-bound to the results. Rule [*FV-Timeout*] computes the effects from e_1 and e_2 using the starting state $\{\pi, \epsilon\}$. The final state is an union of possible effects with corresponding time bounds and arithmetic constraints. Note that, $hd(\Theta_1)$ and $tl(\Theta_1)$ return the event *head* (cf. Definition 6), and the tail of Θ_1 respectively.

$$\frac{\theta' = \theta \cdot (\epsilon\mathtt{\#}t) \quad (t\ \textit{is fresh})}{\mathcal{S} \vdash \{\pi, \theta\}\ \mathtt{delay}[v]\ \{\pi \wedge (t{=}v), \theta'\}}\ [\textit{FV-Delay}] \qquad \frac{\mathcal{S} \vdash \{\pi, \epsilon\}\ e\ \{\Pi_1, \Theta_1\} \quad (t\ \textit{is fresh})}{\mathcal{S} \vdash \{\pi, \theta\}\ \mathtt{deadline}[v]\ e\ \{\Pi_1 \wedge (t{\leq}v), \theta \cdot (\Theta_1\mathtt{\#}t)\}}\ [\textit{FV-Deadline}]$$

$$\frac{\mathcal{S} \vdash \{\pi, \epsilon\}\ e_1\ \{\Pi_1, \Theta_1\} \qquad \mathcal{S} \vdash \{\pi, \epsilon\}\ e_2\ \{\Pi_2, \Theta_2\} \qquad (t_1, t_2\ \textit{are fresh}) \quad \{\Pi_f, \Theta_f\} = \{\Pi_1 \wedge t_1{<}v, (hd(\Theta_1)\mathtt{\#}t_1) \cdot tl(\Theta_1)\} \cup \{\Pi_2 \wedge t_2{=}v, (\epsilon\mathtt{\#}t_2) \cdot \Theta_2\}}{\mathcal{S} \vdash \{\pi, \theta\}\ e_1\ \mathtt{timeout}[v]\ e_2\ \{\Pi_f, \theta \cdot \Theta_f\}}\ [\textit{FV-Timeout}]$$

$$\frac{\mathcal{S} \vdash \{\pi, \epsilon\}\ e_1\ \{\Pi, \Theta\} \qquad \Delta = \bigcup_{i=0}^{|\{\Pi,\Theta\}|-1} \aleph_{Interleave}^{Interrupt(v,\pi_i)}(\theta_i, \epsilon) \qquad \mathcal{S} \vdash \{\Delta\}\ e_2\ \{\Pi', \Theta'\}}{\mathcal{S} \vdash \{\pi, \theta\}\ e_1\ \mathtt{interrupt}[v]\ e_2\ \{\Pi', \theta \cdot \Theta'\}}\ [\textit{FV-Interrupt}]$$

$|\{\Pi, \Theta\}|$ is the size of $\{\Pi, \Theta\}$, i.e., the count of conditioned event sequence $\pi \wedge \theta$.

[*FV-Interrupt*] computes the interruption interleaves of e_1's effects, which come from the over-approximation of all the possibilities. For example, for trace $\mathtt{A} \cdot \mathtt{B}$, the interruption with time t creates three possibilities: $(\epsilon \# t) \vee (\mathtt{A} \# t) \vee ((\mathtt{A} \cdot \mathtt{B}) \# t)$. Then the rule continues to compute the effects of e_2; lastly, it prepends the original history θ to the final results. Algorithm 1 presents the interleaving algorithm for *interruptions*, where + unions program states (cf. Definition 7 and Definition 8 for *fst* and D functions).

Algorithm 1: Interruption Interleaving

Input: $v, \pi, \theta, \theta_{his}$
Output: Program States: Δ
1 **function** $\aleph_{Interleave}^{Interrupt(v,\pi)}(\theta, \theta_{his})$
2 $\Delta \leftarrow []$
3 **foreach** $f \in fst_\pi(\theta)$ **do**
4 $\phi \leftarrow \pi \wedge (t < v) \wedge (\theta_{his} \# t)$
5 $\theta' \leftarrow D_f^\pi(\theta)$
6 $\theta'_{his} \leftarrow \theta_{his} \cdot f$
7 $\Delta' \leftarrow \aleph_{Interleave}^{Interrupt(v,\pi)}(\theta', \theta'_{his})$
8 $\Delta \leftarrow \Delta + \phi + \Delta'$
  return Δ

Theorem 1 (Soundness of Forward Rules). *Given any system configuration* $\zeta = (\mathcal{S}, e)$*, by applying the operational semantics rules, if* $(\mathcal{S}, e) \rightarrow^* (\mathcal{S}', v)$ *has execution time* d *and produces event sequence* φ*; and for any history effect* $\pi \wedge \theta$*, such that* $d_1, \mathcal{S}, \varphi_1 \models (\pi \wedge \theta)$*, and the forward verifier reasons* $\mathcal{S} \vdash \{\pi, \theta\} e \{\Pi, \Theta\}$*, then* $\exists (\pi' \wedge \theta') \in \{\Pi, \Theta\}$ *such that* $(d_1 + d), \mathcal{S}', (\varphi_1 +\!+ \varphi) \models (\pi' \wedge \theta')$*.* ($\zeta \rightarrow^\cup \zeta'$ *denotes the reflexive, transitive closure of* $\zeta \rightarrow \zeta'$*.)*

Proof. See the technical report [16].

5 Temporal Verification via a TRS

The TRS is an automated entailment checker to prove language inclusions between *TimEffs*. It is triggered prior to function calls for the precondition checking; and by the end of verifying a function, for the post condition checking.

Given two effects Φ_1 and Φ_2, the TRS decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid. During the effects rewriting process, the inclusions are in the form of $\Gamma \vdash \Phi_1 \sqsubseteq^\Phi \Phi_2$, a shorthand for: $\Gamma \vdash \Phi \cdot \Phi_1 \sqsubseteq \Phi \cdot \Phi_2$. To prove such inclusions is to check whether all the possible timed traces in the antecedent Φ_1 are legitimately allowed in the timed traces described by the consequent Φ_2. Here Γ is the proof context, i.e., a set of effects inclusion hypothesis; and Φ is the history effects from the antecedent that have been used to match the effects from the consequent. The checking is initially invoked with $\Gamma = \emptyset$ and $\Phi = True \wedge \epsilon$.

Effects Disjunctions. An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent. An inclusion with a disjunctive consequent succeeds if the antecedent entails either of the disjunctions.

$$\frac{\Gamma \vdash \Phi_1 \sqsubseteq \Phi \qquad \Gamma \vdash \Phi_2 \sqsubseteq \Phi}{\Gamma \vdash \Phi_1 \vee \Phi_2 \sqsubseteq \Phi}\ [LHS\text{-}OR] \qquad \frac{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \quad or \quad \Gamma \vdash \Phi \sqsubseteq \Phi_2}{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \vee \Phi_2}\ [RHS\text{-}OR]$$

Now, the inclusions are disjunction-free formulas. Next we provide the definitions and key implementations of auxiliary functions Nullable, First and Derivative. Intuitively, the Nullable function $\delta_\pi(\theta)$ returns a Boolean value indicating

whether $\pi \wedge \theta$ contains the empty trace; the First function $fst_\pi(\theta)$ computes a set of initial *heads*, denoted as h, of $\pi \wedge \theta$; the Derivative function $D_h^\pi(\theta)$ computes a next-state effects after eliminating the head h from the current effects $\pi \wedge \theta$.

Definition 5 (Nullable [7]). *Given any* $\Phi = \pi \wedge \theta$, $\delta_\pi(\theta) : bool = \begin{cases} true & if \ \epsilon \in [\![\pi \wedge \theta]\!] \\ false & if \ \epsilon \notin [\![\pi \wedge \theta]\!] \end{cases}$

$$\delta_\pi(\bot) = \delta_\pi(ev) = false \quad \delta_\pi(\epsilon) = \delta(\theta^\star) = true \quad \delta_\pi(\pi'?\theta) = \delta_\pi(\theta) \quad \delta_\pi(\theta_1 \vee \theta_2) = \delta(\theta_1) \vee \delta(\theta_2)$$
$$\delta_\pi(\theta \cdot \theta_2) = \delta(\theta_1) \wedge \delta(\theta_2) \quad \delta_\pi(\theta_1 || \theta_2) = \delta(\theta_1) \wedge \delta(\theta_2) \quad \delta_\pi(\theta \# t) = SAT(\pi \wedge (t{=}0)) \wedge \delta_\pi(\theta)$$

Definition 6 (Heads). *If h is a head of $\pi \wedge \theta$, then there exist π' and θ', such that $\pi \wedge \theta = \pi' \wedge (h \cdot \theta')$. A head can be t, denoting a pure time passing; $A(v, \alpha^\cup)$, denoting an instant event passing; or $(A(v, \alpha^\cup), t)$, denoting an event passing which takes time t.*

Definition 7 (First). *Given any $\Phi = \pi \wedge \theta$, $fst_\pi(\theta)$ returns a set of heads, be the set of initial elements derivable from effects $\pi \wedge \theta$, where (t' is fresh):*

$$fst_\pi(\bot) = fst_\pi(\epsilon) = \{\} \quad fst_\pi(A(v, \alpha^\cup)) = \{A(v, \alpha^\cup)\} \quad fst_\pi(\epsilon \# t) = \{t\} \quad fst_\pi(\theta^\star) = fst_\pi(\theta)$$
$$fst_\pi(\theta \# t) = \{(A(v, \alpha^\cup), t') \mid A(v, \alpha^\cup) \in fst_\pi(\theta)\} \quad fst_\pi(\theta_1 \vee \theta_2) = fst_\pi(\theta_1) \cup fst_\pi(\theta_2)$$
$$fst_\pi(\pi'?\theta) = fst_\pi(\theta) \qquad fst_\pi(\theta_1 || \theta_2) = fst_\pi(\theta_1) \cup fst_\pi(\theta_2)$$
$$fst_\pi(\theta_1 \cdot \theta_2) = \begin{cases} fst_\pi(\theta_1) \cup fst_\pi(\theta_2) & if \ \ \delta(\theta_1) = true \\ fst_\pi(\theta_1) & if \ \ \delta(\theta_1) = false \end{cases}$$

Definition 8 (*TimEffs* Partial Derivative). *Given any $\Phi = \pi \wedge \theta$, the partial derivative $D_h^\pi(\theta)$ computes the effects for the left quotient $h^{-1}(\pi \wedge \theta)$, cf. Definition 1.*

$$D_h^\pi(\bot) = D_h^\pi(\epsilon) = False \wedge \bot \quad D_h^\pi(A(v, \alpha^\cup)) = (\pi \wedge (h{=}A(v, \alpha^\cup))) \wedge \epsilon \quad D_h^\pi(\theta^\star) = D_h^\pi(\theta) \cdot \theta^\star$$
$$D^\pi_{\tau(\pi_1)}(\pi'?\theta) = \begin{cases} \pi \wedge \pi'?\theta & if \ \ \pi_1 \not\Rightarrow \pi' \\ \pi \wedge \theta & if \ \ \pi_1 \Rightarrow \pi' \end{cases} \quad D_h^\pi(\theta_1 \cdot \theta_2) = \begin{cases} D_h^\pi(\theta_1) \cdot \theta_2 \vee D_h^\pi(\theta_2) & if \ \ \delta_\pi(\theta_1) = true \\ D_h^\pi(\theta_1) \cdot \theta_2 & if \ \ \delta_\pi(\theta_1) = false \end{cases}$$
$$D^\pi_{(A(v,\alpha^*),t)}(\theta) = \bigvee \{D^{\pi'}_{A(v,\alpha^*)}(\theta') \mid (\pi' \wedge \theta') \in D_t^\pi(\theta)\}$$
$$D_t^\pi(\theta \# t') = (\pi \wedge t + t'' = t') \wedge \theta \# t'' \quad (t'' \ is \ fresh) \qquad D_h^\pi(\theta_1 \vee \theta_2) = D_h^\pi(\theta_1) \vee D_h^\pi(\theta_2)$$
$$D^\pi_{A(v,\alpha^*)}(\theta \# t) = \bigvee \{(\pi' \wedge (\theta' \# t)) \mid (\pi' \wedge \theta') \in D^\pi_{A(v,\alpha^*)}(\theta)\} \quad D_h^\pi(\theta_1 || \theta_2) = \bar{\bar{D}}_h^\pi(\theta_1) || \bar{\bar{D}}_h^\pi(\theta_2)$$

Notice that the derivatives of a parallel composition makes use of the *Parallel Derivative* $\bar{\bar{D}}_h^\pi(\theta)$, defined as follows: $\bar{\bar{D}}_h^\pi(\theta) = \begin{cases} \pi \wedge \theta & if \ \ D_h^\pi(\pi \wedge \theta) = (False \wedge \bot) \\ D_h^\pi(\theta) & otherwise \end{cases}$

5.1 Rewriting Rules. Given the well-defined auxiliary functions above, we now discuss the key rewriting rules that deployed in effects inclusion proofs.

$$\frac{}{\Gamma \vdash \pi \wedge \bot \sqsubseteq \Phi} \ [\texttt{Bot-LHS}] \qquad \frac{\Phi \neq \pi \wedge \bot}{\Gamma \vdash \Phi \not\sqsubseteq \pi \wedge \bot} \ [\texttt{Bot-RHS}]$$

$$\frac{\delta_{\pi_1}(\theta_1) \wedge \neg \delta_{\pi_2}(\theta_2)}{\Gamma \vdash \pi_1 \wedge \theta_1 \not\sqsubseteq \pi_2 \wedge \theta_2} \ [\texttt{DISPROVE}] \qquad \frac{\pi_1 \Rightarrow \pi_2 \qquad fst_{\pi_1}(\theta_1) = \{\}}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2} \ [\texttt{PROVE}]$$

[7] $SAT(\pi)$ stands for querying the Z3 theorem prover to check the satisfiability of π.

Axiom rules [Bot-LHS] and [Bot-RHS] are analogous to the standard propositional logic, $\bot$ (referring to *false*) entails any effects, while no *non-false* effects entails $\bot$. [DISPROVE] is used to disprove the inclusions when the antecedent is nullable, while the consequent is not nullable.

We use two rules to prove an inclusion: (i) [PROVE] is used when the antecedent has no head; and (ii) [REOCCUR] proves an inclusion when there exist inclusion hypotheses in the proof context Γ, which are able to soundly prove the current goal. [UNFOLD] is the inductive step of unfolding the inclusions. The proof of the original inclusion succeeds if all the derivative inclusions succeed.

$$\frac{(\pi_1 \wedge \theta_1 \sqsubseteq \pi_3 \wedge \theta_3) \in \Gamma \qquad (\pi_3 \wedge \theta_3 \sqsubseteq \pi_4 \wedge \theta_4) \in \Gamma \qquad (\pi_4 \wedge \theta_4 \sqsubseteq \pi_2 \wedge \theta_2) \in \Gamma}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\texttt{REOCCUR}]$$

$$\frac{H = fst_{\pi_1}(\theta_1) \qquad \Gamma' = \Gamma, (\pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2) \qquad \forall h \in H.\ (\Gamma' \vdash D_h^{\pi_1}(\theta_1) \sqsubseteq D_h^{\pi_2}(\theta_2))}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\texttt{UNFOLD}]$$

Theorem 2 (Termination of the TRS). *The TRS is terminating.*

Proof. See the technical report [16].

Theorem 3 (Soundness of the TRS). *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns TRUE with a proof, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. See the technical report [16].

6 Implementation and Evaluation

To show the feasibility, we prototype our automated verification system using OCaml (∼5k LOC); and prove soundness for both the forward verifier and the TRS. We set up two experiments to evaluate our implementation: i) functionality validation via verifying symbolic timed programs; and ii) comparison with PAT [17] and Uppaal [3] using real-life Fischer's mutual exclusion algorithm. Experiments are done on a MacBook with a 2.6 GHz 6-Core Intel i7 processor. The source code and the evaluation benchmark are openly accessible from [18].

6.1 Experimental Results for Symbolic Timed Models. We manually annotate *TimEffs* specifications for a set of synthetic examples (for about 54 programs), to test the main contributions, including: computing effects from symbolic timed programs written in C^t; and the inclusion checking for *TimEffs* with the parallel composition, block waiting operator and shared global variables.

Table 3 presents the evaluation results for another 16 C^t programs[8], and the annotated temporal specifications are in a 1:1 ratio for succeeded/failed cases. The table records: **No.**, index of the program; **LOC**, lines of code; **Forward(ms)**, effects computation time; **#Prop(✓)**, number of valid properties; **Avg-Prove(ms)**, average proving time for the valid properties; **#Prop(✗)**, number of invalid properties; **Avg-Dis(ms)**, average disproving time for the invalid properties; **#AskZ3**, number of querying Z3 through out the experiments.

[8] All programs contain timed constructs, conditionals, and parallel compositions.

Table 3. Experimental Results for Manually Constructed Synthetic Examples.

No.	LOC	Forward(ms)	#Prop(✓)	Avg-Prove(ms)	#Prop(✗)	Avg-Dis(ms)	#AskZ3
1	26	0.006	5	52.379	5	21.31	77
2	37	43.955	5	83.374	5	52.165	188
3	44	32.654	5	52.524	5	33.444	104
4	72	202.181	5	82.922	5	55.971	229
5	98	42.706	7	149.345	7	60.325	396
6	134	403.617	7	160.932	7	292.304	940
7	133	51.492	7	17.901	7	47.643	118
8	173	57.114	7	40.772	7	30.977	128
9	182	872.995	9	252.123	9	113.838	1142
10	210	546.222	9	146.341	9	57.832	570
11	240	643.133	9	146.268	9	69.245	608
12	260	1032.31	9	242.699	9	123.054	928
13	265	12558.05	11	150.999	11	117.288	2465
14	286	12257.834	11	501.994	11	257.800	3090
15	287	1383.034	11	546.064	11	407.952	1489
16	337	49873.835	11	1863.901	11	954.996	15505

Observations: i) the proving/disproving time increases when the effect computation time increases because larger **Forward(ms)** indicates the higher complexity w.r.t the timed constructs, which complicates the inclusion checking; ii) while *the number of querying Z3 per property* (**#AskZ3**/(**#Prop(✓)**+**#Prop(✗)**)) goes up, the proving/disproving time goes up. Besides, we notice that iii) the disproving times for invalid properties are constantly lower than the proving process, regardless of the program's complexity, which is as expected in a TRS.

6.2 Verifying Fischer's mutual exclusion algorithm. As shown in Fig. 4, the data in columns **PAT(s)** and **Uppaal(s)** are drawn from prior work [19], which indicate the time to prove Fischer's mutual exclusion w.r.t the number of processes (**#Proc**) in PAT and Uppaal respectively. For our system, based on the implementation presented in Fig. 5, we are able to prove the mutual exclusion properties, given the arithmetic constraint `d<e`. Besides, the system disproves mutual exclusion when `d`$\leq$`e`. We record the proving (**Prove(s)**) and disproving (**Disprove(s)**) time and their number of uniquely querying Z3 (**#AskZ3-u**).

Table 4. Comparison with PAT via verifying Fischer's mutual exclusion algorithm

#Proc	Prove(s)	#AskZ3-u	Disprove(s)	#AskZ3-u	PAT(s)	Uppaal(s)
2	0.09	31	0.110	37	$\leq$0.05	$\leq$0.09
3	0.21	35	0.093	42	$\leq$0.05	$\leq$0.09
4	0.46	63	0.120	47	0.05	0.09
5	25.0	84	0.128	52	0.15	0.19

Observations: i) automata-based model checkers (both PAT and Uppaal) are vastly efficient when given concrete values for constants `d` and `e`; however ii) our proposal is able to symbolically prove the algorithm by only providing the constraints of `d` and `e`, which cannot be achieved by existing model checkers; ii) our verification time largely depends on the number of querying Z3, which is optimized in our implementation by keeping a table for already queried constraints.

6.3 Case Study: Prove it when Reoccur. Termination of TRS is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [14], demonstrated in Table 5. In step ②, in order to eliminate the first event B, $\texttt{A}^\star\texttt{\#tR}$ has to be reduced to ϵ, therefore the RHS time constraint has been strengthened to tR=0. Looking at the sub-tree (I), in step ⑤, tL and tR are split into $\texttt{tL}^1+\texttt{tL}^2$ and $\texttt{tR}^1+\texttt{tR}^2$. Then in step ⑥, $\texttt{A\#tL}^1$ together with $\texttt{A\#tR}^1$ are eliminated, unifying $\texttt{tL}^1$ and $\texttt{tR}^1$ by adding the side constraint $\texttt{tL}^1=\texttt{tR}^1$. In step ⑧, we observe the proposition is isomorphic with one of the the previous step, marked using (‡). Hence we apply the rule [REOCCUR] to prove it with a succeed side constraints entailment.

Table 5. The reoccurrence proving example. (I) is the left hand side sub-tree of the main rewriting proof tree.

$$\dfrac{\dfrac{\dfrac{}{\texttt{tL<3}\wedge(\texttt{A}^\star\texttt{\#tL})\cdot\texttt{B} \sqsubseteq \texttt{tR<4}\wedge(\texttt{A}^\star\texttt{\#tR})\cdot\texttt{B}}\ (I) \qquad \dfrac{\dfrac{\dfrac{}{\texttt{True}\wedge\epsilon\sqsubseteq\texttt{tR=0}\wedge\epsilon}\ ④\ [\texttt{PROVE}]}{\texttt{True}\wedge\cancel{\texttt{B}}\sqsubseteq\texttt{tR=0}\wedge\cancel{\epsilon\cdot\texttt{B}}}\ ③\ [\texttt{Normal}]}{\texttt{True}\wedge\texttt{B}\sqsubseteq\texttt{tR<4}\wedge(\texttt{A}^\star\texttt{\#tR})\cdot\texttt{B}}\ ②\ [\texttt{UNFOLD}]}{(\texttt{tL<3}\wedge(\texttt{A}^\star\texttt{\#tL})\cdot\texttt{B})\vee(\texttt{True}\wedge\texttt{B})\sqsubseteq\texttt{tR<4}\wedge(\texttt{A}^\star\texttt{\#tR})\cdot\texttt{B}}\ ①\ [\texttt{OR-LHS}]$$

(I) :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{tL<3}\wedge\texttt{tL}^1+\texttt{tL}^2=\texttt{tL}\wedge\texttt{tR}=\texttt{tR}^1+\texttt{tR}^2\wedge\texttt{tL}^1=\texttt{tR}^1\wedge\texttt{tL}^2=\texttt{tR}^2\Rightarrow\texttt{tR<4}}\ ⑧\ [\texttt{REOCCUR}]}{\texttt{tL<3}\wedge(\texttt{A}^\star\texttt{\#tL}^2)\cdot\texttt{B}\sqsubseteq\texttt{tR<4}\wedge(\texttt{A}^\star\texttt{\#tR}^2)\cdot\texttt{B}\quad(‡)}\ ⑦\ [\texttt{UNFOLD}]}{\texttt{tL<3}\wedge\cancel{\texttt{A\#tL}^1}\cdot\texttt{A}^\star\texttt{\#tL}^2\cdot\texttt{B}\sqsubseteq\texttt{tR<4}\wedge\cancel{\texttt{A\#tR}^1}\cdot\texttt{A}^\star\texttt{\#tR}^2\cdot\texttt{B}}\ ⑥\ [\texttt{UNFOLD}]\ \pi_u{:}\texttt{tL}^1=\texttt{tR}^1}{\texttt{tL<3}\wedge(\texttt{A\#tL}^1\cdot\texttt{A}^\star\texttt{\#tL}^2)\cdot\texttt{B}\sqsubseteq\texttt{tR<4}\wedge(\texttt{A\#tR}^1\cdot\texttt{A}^\star\texttt{\#tR}^2)\cdot\texttt{B}}\ ⑤\ [\texttt{SPLIT}]\texttt{tL}^1+\texttt{tL}^2=\texttt{tL}\wedge\texttt{tR}^1+\texttt{tR}^2=\texttt{tR}$$

$$\texttt{tL<3}\wedge(\texttt{A}^\star\texttt{\#tL})\cdot\texttt{B}\sqsubseteq\texttt{tR<4}\wedge(\texttt{A}^\star\texttt{\#tR})\cdot\texttt{B}\quad(‡)$$

6.4 Discussion. Our implementation is the first that proves the inclusion of symbolic TAs, which is considered significant because it overcomes the following main limitations of traditional timed model checking: i) TAs cannot be used to specify/verify incompletely specified systems (i.e., whose timing constants have yet to be known) and hence cannot be used in early design phases; ii) verifying a system with a set of timing constants usually requires enumerating all of them if they are supposed to be integer-valued; iii) TAs cannot be used to verify systems with timing constants to be taken in a real-valued dense interval.

7 Related Work

7.1 Verification Framework. This work draws the most similarities to [20], which also deploys a forward verifier and a TRS for extended regular expressions. The differences are: i) [20] targets general-purpose sequential programs without shared variables, whereas this work targets time-critical programs with the presence of concurrency and global shared states; ii) the dependent values in [20] denote the number of repetitions of a trace, whereas in this work, they abstract the real-time bounds; iii) in this work, the TRS supports inclusion checking for the block waiting operator π? and the concurrent composition ||. These are essential in timed verification (or, more generally, for distributed systems), which are not supported in [20] or any other TRS-related works.

7.2 Specifications and Real-Time Verification. Apart from compositional modelling for real-time systems based on timed-process algebras, such as Timed CSP [8] and CCS+Time [21], there have been a number of translation-based approaches on building verification support for timed-process algebras. For example, in [8], Timed CSP is translated to TAs (TAs) so that the model checker Uppaal [3] can be applied. On the other hand, all the translation-based approaches share the common problem: the overhead introduced by the complex translation makes it particularly inefficient when *disproving* properties. We are of the opinion that in that the goal of verifying real-time systems, in particular safety-critical systems is to check logical temporal properties, which can be done without constructing the whole reachability graph or the full power of model-checking. We consider our approach is simpler as it is based directly on constraint-solving techniques and can be fairly efficient in verifying systems consisting of many components as it avoids to explore the whole state-space [20,22].

This work draws similarities to Real-Time Maude [23], which complements timed automata with more expressive object-oriented specifications.

7.3 Clock Manipulation and Zone-based Bisimulation. The concept of implicit clocks has also been used in time Petri nets, and implemented in a several model checking engines, e.g., [24]. On the other hand, to make model checking more efficient with *explicit* clocks, [25,26,27,28] work on dynamically deleting or merging clocks. Our work also draw connections with region/zone-based bisimulations [29], which is broadly used in reasoning timed automata.

8 Conclusion

This work provides an alternative approach for verifying real-time systems, where temporal behaviors are reasoned at the source level, and the specification expressiveness goes beyond traditional Timed Automata. We define the novel effects logic *TimEffs*, to capture real-time behavioral patterns and temporal properties. We demonstrate how to build axiomatic semantics (or rather an effects system) for C^t via timed-trace processing functions. We use this semantic model to enable a Hoare-style forward verifier, which computes the program effects constructively. We present an effects inclusion checker – the TRS – to efficiently prove the annotated temporal properties. We prototype the verification system and show its feasibility. To the best of our knowledge, our work proposes the first algebraic TRS for solving inclusion relations between timed specifications.

Limitations And Future Work. Our TRS is incomplete, meaning there exist valid inclusions which will be disproved in our system. That is mainly because of insufficient unification in favour of achieving automation. We also foresee the possibilities of adding other logics into our existing trace-based temporal logic, such as separation logic for verifying heap-manipulating distributed programs.

9 Acknowledgements

The authors would like to thank anonymous reviewers for their comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOET32021-0001.

References

1. R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: https://doi.org/10.1016/0304-3975(94)90010-8
2. X. Wang, J. Sun, T. Wang, and S. Qin, "Language inclusion checking of timed automata with non-zenoness," *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 995–1008, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2017.2653778
3. K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 134–152, 1997. [Online]. Available: https://doi.org/10.1007/s100090050010
4. S. Yovine, "KRONOS: A verification tool for real-time systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 123–133, 1997. [Online]. Available: https://doi.org/10.1007/s100090050009
5. F. Wang, R. Wu, and G. Huang, "Verifying timed and linear hybrid rule-systems with RED," in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005), Taipei, Taiwan, Republic of China, July 14-16, 2005*, W. C. Chu, N. J. Juzgado, and W. E. Wong, Eds., 2005, pp. 448–454.
6. K. Havelund, A. Skou, K. G. Larsen, and K. Lund, "Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL," in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*. IEEE Computer Society, 1997, pp. 2–13. [Online]. Available: https://doi.org/10.1109/REAL.1997.641264
7. K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: an industrial case study," in *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, W. H. Wolf, Ed. ACM, 2005, pp. 299–306. [Online]. Available: https://doi.org/10.1145/1086228.1086283
8. J. S. Dong, P. Hao, S. Qin, J. Sun, and W. Yi, "Timed automata patterns," *IEEE Trans. Software Eng.*, vol. 34, no. 6, pp. 844–859, 2008. [Online]. Available: https://doi.org/10.1109/TSE.2008.52
9. A. David and M. D. Möller, "From huppaal to uppaal–a translation from hierarchical timed automata to flat timed automata," 2001.
10. L. Lamport, "Real-time model checking is really simple," in *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, ser. Lecture Notes in Computer Science, D. Borrione and W. J. Paul, Eds., vol. 3725. Springer, 2005, pp. 162–175. [Online]. Available: https://doi.org/10.1007/11560548_14
11. P. L. P. Ltd., https://www.programiz.com/javascript/setTimeout, 2022.
12. V. M. Antimirov and P. D. Mosses, "Rewriting extended regular expressions," *Theor. Comput. Sci.*, vol. 143, no. 1, pp. 51–72, 1995. [Online]. Available: https://doi.org/10.1016/0304-3975(95)80024-4
13. V. Antimirov, "Partial derivatives of regular expressions and finite automata constructions," in *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 1995, pp. 455–466.
14. J. Brotherston, "Cyclic proofs for first-order logic with inductive definitions," in *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005,*

Proceedings, ser. Lecture Notes in Computer Science, B. Beckert, Ed., vol. 3702. Springer, 2005, pp. 78–92. [Online]. Available: https://doi.org/10.1007/11554554_8
15. L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24
16. Anonymous, https://www.comp.nus.edu.sg/~yahuis/TACAS2023.pdf, 2023.
17. J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: towards flexible verification under fairness," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 709–714. [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4_59
18. Y. Song, https://zenodo.org/record/7192718#.Y7rTmi8RpOQ, 2022.
19. Y. Liu, J. Sun, and J. S. Dong, "PAT 3: An extensible architecture for building multi-domain model checkers," in *IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011*, T. Dohi and B. Cukic, Eds. IEEE Computer Society, 2011, pp. 190–199. [Online]. Available: https://doi.org/10.1109/ISSRE.2011.19
20. Y. Song and W. Chin, "Automated temporal verification of integrated dependent effects," in *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, ser. Lecture Notes in Computer Science, S. Lin, Z. Hou, and B. P. Mahony, Eds., vol. 12531. Springer, 2020, pp. 73–90. [Online]. Available: https://doi.org/10.1007/978-3-030-63406-3_5
21. W. Yi, "CCS + time = an interleaving model for real time systems," in *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, Eds., vol. 510. Springer, 1991, pp. 217–228. [Online]. Available: https://doi.org/10.1007/3-540-54233-7_136
22. W. Yi, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving," in *Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994*, ser. IFIP Conference Proceedings, D. Hogrefe and S. Leue, Eds., vol. 6. Chapman & Hall, 1994, pp. 243–258.
23. P. C. Ölveczky and J. Meseguer, "Semantics and pragmatics of Real-Time Maude," *Higher-Order and Symbolic Computation*, vol. 20, no. 1-2, pp. 161–196, 2007.
24. B. Berthomieu and F. Vernadat, "Time petri nets analysis with TINA," in *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*. IEEE Computer Society, 2006, pp. 123–124. [Online]. Available: https://doi.org/10.1109/QEST.2006.56
25. C. Daws and S. Yovine, "Reducing the number of clock variables of timed automata," in *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*. IEEE Computer Society, 1996, pp. 73–81. [Online]. Available: https://doi.org/10.1109/REAL.1996.563702
26. S. Balaguer and T. Chatain, "Avoiding shared clocks in networks of timed automata," *Log. Methods Comput. Sci.*, vol. 9, no. 4, 2013. [Online]. Available: https://doi.org/10.2168/LMCS-9(4:13)2013

27. M. Muñiz, B. Westphal, and A. Podelski, "Detecting quasi-equal clocks in timed automata," in *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, ser. Lecture Notes in Computer Science, V. A. Braberman and L. Fribourg, Eds., vol. 8053. Springer, 2013, pp. 198–212. [Online]. Available: https://doi.org/10.1007/978-3-642-40229-6_14
28. S. Guha, C. Narayan, and S. Arun-Kumar, "Reducing clocks in timed automata while preserving bisimulation," in *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, ser. Lecture Notes in Computer Science, P. Baldan and D. Gorla, Eds., vol. 8704. Springer, 2014, pp. 527–543. [Online]. Available: https://doi.org/10.1007/978-3-662-44584-6_36
29. L. Luthmann, H. Göttmann, and M. Lochau, "Checking timed bisimulation with bounded zone-history graphs - technical report," *CoRR*, vol. abs/1910.08992, 2019. [Online]. Available: http://arxiv.org/abs/1910.08992

Parameterized Verification under TSO with Data Types

Parosh Aziz Abdulla[1], Mohamad Faouzi Atig[1], Florian Furbach[1](✉), Adwait A. Godbole[3], Yacoub G. Hendi[1], Shankara N. Krishna[2], and Stephan Spengler[1]

[1] Uppsala University, Uppsala, Sweden
florian.furbach@it.uu.se
[2] Indian Institute of Technology Bombay, Mumbai, India
[3] UC Berkeley, Berkeley, USA

We consider parameterized verification of systems executing according to the total store ordering (TSO) semantics. The processes manipulate abstract data types over potentially infinite domains. We present a framework that translates the reachability problem for such systems to the reachability problem for register machines enriched with the given abstract data type. We use the translation to obtain tight complexity bounds for TSO-based parameterized verification over several abstract data types, such as push-down automata, ordered multi push-down automata, one-counter nets, one-counter automata, and Petri nets. We apply the framework to get complexity bounds for higher order stack and counter variants as well.

1 Introduction

A *parameterized system* consists of a fixed but arbitrary number of identical processes that execute in parallel. The goal of *parameterized verification* is to prove the correctness of the system regardless of the number of processes. Examples for such systems are sensor networks, leader election protocols, and mutual exclusion protocols. The topic has been the subject of intensive research for more than three decades (see e.g. [10,32,13,6]), and it is the subject of one chapter of the Handbook of Model Checking [8]. Research on parameterized verification has been mostly conducted under the premise that (i) the processes run according to the classical Sequential Consistency (SC) semantics, and (ii) the processes are finite-state machines.

Under SC, the processes operate on a set of shared variables through which they communicate *atomically*, i.e., read and write operations take effect immediately. In particular, a write operation is visible to all the processes as soon as the writing process carries out its operation. Therefore, the processes always maintain a uniform view of the shared memory: they all see the latest value written on any given variable, hence we can interpret program runs as interleavings of sequential process executions. Although SC has been immensely popular as an intuitive way of understanding the behaviours of concurrent processes, it is not realistic to assume computation platforms guarantee SC anymore. The reason is that, due to hardware and compiler optimizations, most modern platforms

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 588–606, 2023.
https://doi.org/10.1007/978-3-031-30823-9_30

allow more relaxed program behaviours than those permitted under SC, leading to so-called *weak memory models*. Weakly consistent platforms are found at all levels of system design such as multiprocessor architectures (e.g., [48,47]), Cache protocols (e.g., [46,21]), language level concurrency (e.g., [41]), and distributed data stores (e.g., [17]). Therefore, in recent years, research on the parameterized verification of concurrent programs under weak memory models have started to become popular. Notable examples are the cases of the TSO semantics [4] and the Release-Acquire semantics of C11 [39].

In a parallel development, several works have extended the basic model of parameterized systems (under the SC semantics) by considering processes that are infinite-state systems. The most dominant such class has been the case where the individual processes are variants of push-down automata [36,33,28,28,40,42,30]

Parameterized verification is difficult, even under the original assumption of both SC and finite-state processes as we still need to handle an infinite state space. The extension to weakly consistent systems is even more complex due to the intricate extra process behaviours. Almost all weak memory models induce infinite state spaces even without parameterization and even when the program itself is finite-state. Therefore, performing parameterized verification under weak consistency requires handling a state space that is infinite in two dimensions; one due to parameterization and one due to the weak memory model. The same applies to the extension of parameterized verification under SC where the processes are infinite-state: in addition to infiniteness due to parameterization, we have a second source of infinity due to the infiniteness of the processes.

In this paper, we combine the above two extensions. We study parameterized verification of programs under the TSO semantics, where the processes use infinite data structures such as stacks and counters. The framework is uniform in that the manipulation can be described using an abstract data type.

We revisit the pivot abstraction technique presented in [4]. As a first contribution, we show that we can capture pivot abstraction precisely, using a class of register machines in which the registers assume values over a finite domain. We show that, for any given abstract data type A, we can reduce, in polynomial time, the parameterized verification problem under TSO and A to the reachability problem for register machines manipulating A. Furthermore, we show that the reduction also holds in the other direction: the reachability problem for register machines over A is polynomial-time reducible to the parameterized verification problem under TSO for A. In particular, the model abstracts away the semantics of TSO (in fact, it abstracts away concurrency altogether) since we are dealing with a single register machine.

We summarize the contributions of the paper as follows:

- We present a register abstraction scheme that captures the behaviour of parameterized systems under the TSO semantics.
- We translate parameterized verification under the TSO semantics when the processes manipulate an ADT A, to the reachability problem for register machines operating over A.
- We instantiate the framework for deciding the complexity of parameterized verification under TSO for different abstract data types. In particular we

show the problem is PSPACE-complete when A is a one-counter, EXPTIME-complete if A is a stack, 2-ETIME-complete if A is an ordered multi stack, and EXPSPACE-complete if A is a Petri net. We obtain further complexity bounds for higher order counter and stacks.

Related Work There has been an extensive research effort on parameterized verification since the 1980s (see [13,8] for recent surveys of the field). Early works showed the undecidability of the general problem (even assuming finite-state processes) [10], and hence the emphasis has been on finding useful special cases. Such cases are characterized by three aspects, namely the system topology (unordered, arrays, trees, graphs, rings, etc.), the allowed communication patterns (shared memory, Rendez-vous, broadcast, lossy channels, etc.), and the process types (anonymous, with IDs, with priorities, etc.) [27,20,31,24,23,43].

Another line of research to counter undecidability are over-approximations based on regular model checking [38,14,16,1], monotonic abstraction [5], and symmetry reduction [37,22,7].

A seminal work in the area is the paper by German and Sistla [32]. The authors consider the verification of systems consisting of an arbitrary number of finite-state processes interacting through Rendez-Vous communication. The paper shows that the model checking problem is EXPSPACE-complete. In a series of more recent papers, parameterized verification has been considered in the case where the individual processes are push-down automata. [36,33,28,40,42,30].All the above works assume the SC semantics.

Due to the relevance of weak memory models in parameterized verification, papers on the topic have started to appear in the last two years. The paper [4] considers parameterized verification of programs running under TSO, and shows that the reachability problem is PSPACE-complete. However, the paper assumes that the processes are finite-state and, in particular, the processes do not manipulate unbounded data domains. The model of the paper corresponds to the particular case of our framework where we take the abstract data type to be empty. In this case our framework also implies PSPACE-completeness.

The paper [39] shows PSPACE-completeness when the underlying semantics is the Release-Acquire fragment of C11. The latter semantics gives rise to different semantics compared to TSO. The paper also considers finite-state processes.

The paper [2] considers parameterized verification of programs running under TSO. However, the paper applies the framework of well-structured systems where the buffers of the processes are modeled as lossy channels, and hence the complexity of the algorithm is non-primitive recursive. In particular, the paper does not give any complexity bounds for the reachability problem (or any other verification problems). Conchon et al. [19] address the parameterized verification of programs under TSO as well. They make use of Model Checker Modulo Theories, no decidability or complexity results are given. The paper [15] considers checking the robustness property against SC for parameterized systems running under the TSO semantics. However, the robustness problem is entirely different from reachability and the techniques and results developed in the paper cannot

be applied in our setting. The paper shows that the problem is ExpSpace-hard. All these works assume finite-state processes.

In contrast to all the above works, the current paper is the first paper that studies decidability and complexity of parameterized verification under the TSO semantics when the individual processes are infinite-state.

2 Preliminaries

We denote a function f between sets A and B by $f : A \longrightarrow B$. We write $f[a \leftarrow b]$ to denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ for all $x \neq a$.

For a finite set A, we use $|A|$ to refer to the size of A. We also use A^* to denote the set of words over A including the empty word ϵ. For a word $w \in A^*$, we use $|w|$ to refer to the length of w. We say a word w is *differentiated* if all symbols in w are pairwise different. The set A^{diff} is the set of all differentiated words over the set A. Finally, for a differentiated word w, we define $\mathrm{pos}(w)(a)$ as the unique position of the letter a in w.

A *labelled transition system* is a tuple $\langle \mathsf{C}, \mathsf{C}_{\mathsf{init}}, \mathsf{Labs}, \longrightarrow \rangle$, where C is the set of configurations, $\mathsf{C}_{\mathsf{init}} \subseteq \mathsf{C}$ is the set of initial configurations, Labs is a finite set of labels and $\longrightarrow \subseteq \mathsf{C} \times \mathsf{Labs} \times \mathsf{C}$ is the transition relation over the set of configurations. For a transition $\langle \mathsf{c}_1, \mathsf{lab}, \mathsf{c}_2 \rangle \in \longrightarrow$, we usually write $\mathsf{c}_1 \xrightarrow{\mathsf{lab}} \mathsf{c}_2$ instead. We use $\mathsf{c}_1 \longrightarrow \mathsf{c}_2$ to denote that $\mathsf{c}_1 \xrightarrow{\mathsf{lab}} \mathsf{c}_2$ for some $\mathsf{lab} \in \mathsf{Labs}$. Furthermore, we write $\xrightarrow{*}$ to denote the transitive reflexive closure over $\longrightarrow$, and if $\mathsf{c}_1 \xrightarrow{*} \mathsf{c}_2$ then we say c_2 is *reachable from* c_1. If $\mathsf{c}_1 \in \mathsf{C}_{\mathsf{init}}$, then we just say that c_2 is *reachable*. A *run* ρ is an alternating sequence of configurations and labels and is expressed as follows: $\mathsf{c}_0 \xrightarrow{\mathsf{lab}_1} \mathsf{c}_1 \xrightarrow{\mathsf{lab}_2} \mathsf{c}_2 \ldots \mathsf{c}_{n-1} \xrightarrow{\mathsf{lab}_n} \mathsf{c}_n$. Given ρ, we write $\mathsf{c}_0 \xrightarrow{n} \mathsf{c}_n$ meaning that c_n is reachable from c_0 by n steps, and we write $\mathsf{c}_0 \xrightarrow{\rho} \mathsf{c}_n$ meaning that c_n is reachable from c_0 through the run ρ.

3 Abstract Data Types (ADT)

In this section, we introduce the notion of abstract data types (ADTs) which will be used extensively in the paper. An ADT is a labelled transition system $\mathsf{A} = \langle \mathsf{Vals}, \{\mathsf{val}_{\mathsf{init}}\}, \mathsf{Ops}, \longrightarrow_{\mathsf{A}} \rangle$. Intuitively, this describes the behaviour of some data type such as a stack, or a counter. Vals is the set of configurations of A. It describes the possible values the data type can assume. The initial configuration is $\mathsf{val}_{\mathsf{init}} \in \mathsf{Vals}$. The set of labels Ops represents the operations that can be executed on the data type and the transition relation $\longrightarrow_{\mathsf{A}} \in \mathsf{Vals} \times \mathsf{Ops} \times \mathsf{Vals}$ describes the semantics of these operations. Below, we give some concrete examples of abstract data types.

Example 1 (Counter). We define a counter, denoted by the ADT Ct, as follows. The set of configurations $\mathsf{Vals}^{\mathrm{Ct}} = \mathbb{N}$ are the natural numbers. The initial value, denoted by $\mathsf{val}^{\mathrm{Ct}}_{\mathsf{init}}$, is 0. The set of operations is $\mathsf{Ops}^{\mathrm{Ct}} = \{\mathtt{inc}, \mathtt{dec}, \mathtt{isZero}\}$. The transition relation $\longrightarrow_{\mathrm{Ct}}$ is as follows: The operations `inc` and `dec` increase or

decrease the value of the counter by one, respectively. The latter operation is only enabled if the value of the counter is non-zero, otherwise it blocks. Finally, the transition `isZero` checks that the value of the counter is zero, i.e. it is only enabled if that condition is true.

Example 2 (Weak Counter). A weak counter differs from a counter in that it cannot be checked for zero. The ADT wCt representing a weak counter is defined as in Example 1, except the operations of wCt are reduced to $\mathsf{Ops}^{\text{wCt}} = \{\texttt{inc}, \texttt{dec}\}$.

Example 3 (Stack). Let Γ be a finite set representing the stack alphabet. A *stack* $\text{St} = \langle \mathsf{Vals}^{\text{St}}, \{\mathsf{val}^{\text{St}}_{\mathsf{init}}\}, \mathsf{Ops}^{\text{St}}, \rightarrow_{\text{St}} \rangle$ on Γ is defined as follows. The configurations of St are $\mathsf{Vals}^{\text{St}} = \Gamma^*$ and the initial configuration is the empty stack $\mathsf{val}^{\text{St}}_{\mathsf{init}} = \varepsilon$. The set of operations is $\mathsf{Ops}^{\text{St}} = \{\texttt{pop}(\gamma), \texttt{push}(\gamma), \texttt{isEmpty} \mid \gamma \in \Gamma\}$. The transition relation is as follows. For every word $w \in \Gamma^*$ and every symbol $\gamma \in \Gamma$, $\texttt{push}(\gamma)$ adds the symbol γ to the top of the stack. Similiarly, the $\texttt{pop}(\gamma)$ operation removes the topmost symbol from the stack. It is only enabled if the topmost symbol on the stack. The `isEmpty` operation does not change the stack, but can only be performed if the stack is the empty word ε.

Example 4 (Petri Nets). Given a Petri net[44], We can define a corresponding ADT Petri that models its semantics. The values are the markings, the operations are the Petri net transitions and the transition relation is given by the input and output vectors of the Petri net transitions.

Higher Order ADTs We extend the ADT St to higher order stacks referred to as n-St. This is done recursively[18,25]. The formal definition is in the full version of our paper [3]. A value of a level n higher order stack n-St is a stack of level $n-1$ stacks. For level 1, it is the standard stack St. The operations for level n are $\mathsf{Ops}^{n\text{-St}} = \{\texttt{pop}(\gamma), \texttt{push}(\gamma), \texttt{pop}_k, \texttt{push}_k, \mid \gamma \in \Gamma, 2 \leq k \leq n\}$. The operations $\texttt{pop}(\gamma)$ and $\texttt{push}(\gamma)$ are recursively applied to the top element in the stack (which consists of a stack that is one level lower) until the level of the top element is 1. Here, they have the standard stack behaviour. Operations $\texttt{pop}_k$ and $\texttt{push}_k$ are recursively applied to the top element until the level of the element is k. Then, a copy of this level k stack is pushed on top of the original.

Since a counter can be seen as a stack with an alphabet of size 1 (and a bottom element $\perp$), we can extend definitions of wCt and Ct to n-wCt and n-Ct in the same way. We add operations $\texttt{inc}_k, \texttt{dec}_k$. All operations are recursively applied to the top counter. For $\texttt{inc}, \texttt{dec}, \texttt{isZero}$, we use standard behaviour once the level is 1. For $\texttt{inc}_k, \texttt{dec}_k$, we copy/remove the top element once the level is k.

Example 5 (Ordered Multi Stack). We extend the stack to a numbered list of n many stacks n-OMSt [12]. A value of n-OMSt consists of list of stacks $\mathsf{val}^{\text{St}}_1 \ldots \mathsf{val}^{\text{St}}_n$. An operation $\mathsf{Ops}^{n\text{-OMSt}} = \{\texttt{isZero}_i, \texttt{pop}_i(\gamma), \texttt{push}_i(\gamma), \mid \gamma \in \Gamma, i \leq n\}$ works on stack number i in the standard way. One additional condition is that the stacks have to be ordered, meaning an operation $\texttt{pop}_i(\gamma)$ is only enabled if the stacks $1 \ldots i-1$ are empty.

4 TSO with an Abstract Data Type : TSO(A)

In this section, we introduce concurrent programs running under $\mathsf{TSO}(\mathsf{A})$ for an ADT $\mathsf{A} = \langle \mathsf{Vals}, \{\mathsf{val}_{\mathsf{init}}\}, \mathsf{Ops}, \longrightarrow_{\mathsf{A}} \rangle$. These programs consist of concurrent processes where the communication between processes is performed using shared memory under the TSO semantics. In addition, each process maintains a local variable of type A.

Syntax of $\mathsf{TSO}(\mathsf{A})$. Let Dom be a finite data domain and Vars be a finite set of shared variables over Dom. Let $\mathsf{d}_{\mathsf{init}} \in \mathsf{Dom}$ be the initial value of the variables. We define the *instruction set* of $\mathsf{TSO}(\mathsf{A})$ as $\mathsf{Instrs} = \{\mathtt{rd}(\mathsf{x},\mathsf{d}), \mathtt{wr}(\mathsf{x},\mathsf{d}) \mid \mathsf{x} \in \mathsf{Vars}, \mathsf{d} \in \mathsf{Dom}\} \cup \{\mathtt{skip}, \mathtt{mf}\}$, which are called *read*, *write*, *skip* and *memory fence*, respectively.

A process is represented by a finite state transition system. It is given by the tuple $\mathsf{Proc} = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$, where Q is a finite set of states, $\mathsf{q}_{\mathsf{init}} \in \mathsf{Q}$ is the initial state, and $\delta \subseteq \mathsf{Q} \times (\mathsf{Instrs} \cup \mathsf{Ops}) \times \mathsf{Q}$ is the transition relation. We call this tuple the *description* of the process. A concurrent program is a tuple of processes $\mathcal{P} = \langle \mathsf{Proc}_\iota \rangle_{\iota \in \mathcal{I}}$, where $\mathcal{I}$ is some finite set of process identifiers. For each $\iota \in \mathcal{I}$ we have $\mathsf{Proc}^\iota = \langle \mathsf{Q}^\iota, \mathsf{q}^\iota_{\mathsf{init}}, \delta^\iota \rangle$.

Semantics of $\mathsf{TSO}(\mathsf{A})$. We describe the semantics of a program $\mathcal{P}$ running under $\mathsf{TSO}(\mathsf{A})$ by a labelled transition system $\mathcal{T}_\mathcal{P} = \langle \mathsf{C}^\mathcal{P}, \mathsf{C}^\mathcal{P}_{\mathsf{init}}, \mathsf{Labs}^\mathcal{P}, \longrightarrow_\mathcal{P} \rangle$. The formal definition is given in [3]. Under $\mathsf{TSO}(\mathsf{A})$, there is an unbounded FIFO buffer of writes between each process and the memory. A configuration $\mathsf{c} \in \mathsf{C}^\mathcal{P}$ of the system consists of the value of each variable in the shared memory as well as for each process: its local state, its value of the ADT, and the content of the corresponding write buffer.

The labelled transitions $\longrightarrow_\mathcal{P}$ are as follows: A local `skip` transition simply updates the state of the corresponding process. An ADT operation additionally updates the ADT value according to ADT behaviour $\longrightarrow_{\mathsf{A}}$. When a process executes a write instruction, the operation is enqueued as a *pending write message* into its buffer. A message msg is an assignment of the form $\mathsf{msg} = \langle \mathsf{x}, \mathsf{d} \rangle$, where $\mathsf{x} \in \mathsf{Vars}$ and $\mathsf{d} \in \mathsf{Dom}$. We denote the set of all messages by $\mathsf{Msgs} = \mathsf{Vars} \times \mathsf{Dom}$. The buffer content for a process is given as a word over Msgs. The messages inside each buffer are moved non-deterministically to the main memory in a FIFO manner. Once a message reaches the memory, it becomes visible to all the other processes. When executing a read instruction on a variable $\mathsf{x} \in \mathsf{Vars}$, the process first checks its buffer for pending write messages on x. If the buffer contains such a message, then it reads the value of the most recent one. If the buffer contains no write messages on x, then the process fetches the value of x from the memory. The *initial configuration* is $\mathsf{c}^\mathcal{P}_{\mathsf{init}}$, where each process is in its initial state, each ADT holds its initial value, each store buffer is empty and the memory holds the initial values of all variables. Note that since FIFO buffer is unbounded, this is an infinite state transition system, even for finite ADT.

A sequence of transitions $\mathsf{c}_0 \xrightarrow{\mathsf{lab}_1}_\mathcal{P} \mathsf{c}_1 \xrightarrow{\mathsf{lab}_2}_\mathcal{P} \mathsf{c}_2 \ldots \mathsf{c}_{n-1} \xrightarrow{\mathsf{lab}_n}_\mathcal{P} \mathsf{c}_n$ where $\mathsf{c}_0 = \mathsf{c}^\mathcal{P}_{\mathsf{init}}$ is the initial configuration and $\mathsf{lab}_i \in \mathsf{Labs}^\mathcal{P}$ is called a run in the $\mathsf{TSO}(\mathsf{A})$ transition system. If there is a run ending in a configuration with state $\mathsf{q}_{\mathsf{final}}$, then we say $\mathsf{q}_{\mathsf{final}}$ is reachable by Proc under $\mathsf{TSO}(\mathsf{A})$.

5 Parameterized Reachability in TSO(A)

In this section, we consider the parameterized TSO setting which allows for an a priori unbounded number of processes with the same process description. We begin by formally introducing the parameterized state reachability problem, and then develop a generic construction that allows us to represent the TSO semantics (except for the ADT) in a finite manner.

The Parameterized State Reachability Problem Intuitively, parameterization allows for an arbitrary number of identical processes. The parameterized state reachability problem for TSO(A) called TSO(A)-P-Reach identifies a family of (standard) reachability problem instances. We want to determine whether we have reachability in some member of the family. We now introduce this formally.

For a given process description Proc, we consider the program instance, $\mathcal{P}^n_{\mathsf{Proc}}$ parameterized by a natural number n as follows. For $\mathcal{I} = \{1, \ldots, n\}$, let $\mathcal{P}^n_{\mathsf{Proc}} = \langle \mathsf{Proc}_1, \ldots, \mathsf{Proc}_n \rangle$ with $\mathsf{Proc}_\iota = \mathsf{Proc}$ for all $\iota \in \mathcal{I}$. That is, the n^{th} *slice* of the parameterized family of programs contains n processes, all with identical descriptions Proc. We require that all processes maintain copies of the ADT A.

TSO(A)-P-Reach:
Given: A process $\mathsf{Proc} = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$, an ADT A, and a state $\mathsf{q}_{\mathsf{final}} \in \mathsf{Q}$,
Decide: Is there a $n \in \mathbb{N}$ s.t. $\mathsf{q}_{\mathsf{final}}$ is reachable by $\mathcal{P}^n_{\mathsf{Proc}}$ under TSO(A)?

When talking about a certain family of ADTs, e.g. the family of petri nets, we write TSO(Petri)-P-Reach and mean the restriction of TSO(A)-P-Reach to petri nets, i.e. to instances where A is a petri net.

The main difference between the non-parameterized case and the parameterized case of the problem is that in the first case the index set $\mathcal{I}$ is *a priori* fixed, while in the second case it can be *arbitrary*. This results in $\mathsf{C}^{\mathcal{P}}_{\mathsf{init}}$ being a singleton in the non-parameterized case while it becomes infinite (one initial state for each n-slice) in the parameterized case.

We determine upper and lower bounds for the complexity of the state reachability problem. The challenge of solving this problem varies with the ADT. This problem for plain TSO without an ADT has been studied in [4]. They showed that the problem can be decided in PSpace and is in fact PSpace-complete. The result is based on an abstraction technique called the *pivot* semantics. The pivot semantics is *exact* in the sense that a state q is reachable under parameterized TSO if and only if it is reachable under the pivot semantics.

We show that the dynamics underlying the pivot abstraction can be generalized to our model with ADT. We show that the pivot abstraction can be extended to obtain a register machine. We use this construction to give a general characterization of TSO(A)-P-Reach. First, we recall the pivot abstraction.
The Pivot Abstraction [4]. For a set of variables Vars and data domain Dom, processes generate pending write messages from the set $\mathsf{Msgs} = \mathsf{Vars} \times \mathsf{Dom}$ by

executing wr instructions. This set has size $|\mathsf{Vars}| \cdot |\mathsf{Dom}|$ and hence at most as many distinct (variable, value) pairs can be produced in any run. For a run ρ of the program, for each message $\mathsf{msg} = \langle \mathsf{x}, \mathsf{d} \rangle \in \mathsf{Msgs}$ we can define the first point along ρ at which some write on variable x with value d is propagated to the memory. The pivot abstraction identifies these points as *pivot* points $\mathrm{pvt}(\mathsf{msg})$, for each distinct message in Msgs. For a write message msg under ρ, the pivot point $\mathrm{pvt}(\mathsf{msg})$ is the first point of propagation of msg to the memory under ρ.

The core observation is that if at some point in ρ, a process Proc_ι propagates a message $\mathsf{msg} = \langle \mathsf{x}, \mathsf{d} \rangle$ from its buffer to the memory, then after that point, the value d will always be available to read on variable x from the shared memory. Technically, this follows from parameterization. There are arbitrarily many processes executing identical descriptions. This means transitions of the original process Proc_ι can be mimicked by a *clone* process $\mathsf{Proc}_{\iota'}$ identical to Proc_ι. Hence, $\mathsf{Proc}_{\iota'}$ can replicate the execution of Proc_ι right up to the point where the message msg is the oldest message in its buffer. Then a single propagate step updates the value of x in the shared memory to d. There can be arbitrarily many such clones and the propagate step can happen at any time. It follows that beyond the $\mathrm{pvt}(\mathsf{msg})$ point in ρ, the value d can always be read from x.

For distinct messages from Msgs, we can order the pivot points corresponding to these messages according to the order in which they appear in ρ. This gives us a *first update sequence*, denoted by ω. No two messages in ω are the same; the set of such sequences is the set of differentiated words $\mathsf{Msgs}_{\mathsf{diff}}$. A message $\mathsf{msg} \in \mathsf{Msgs}$ in ω has the *rank* k if it is the k-th pivot point in ω.

Providers. The pivot abstraction simulates a run ρ under the TSO semantics by running abstract processes called *providers* in a sequential manner. For $1 \leq k \leq |\omega| + 1$, the k-provider simulates the process that generates the write of the rank k message $\langle \mathsf{x}, \mathsf{d} \rangle$ corresponding to the k-pivot in ρ. The k-provider completes its task when it has simulated this process until the point it generates $\langle \mathsf{x}, \mathsf{d} \rangle$. At this point, it invokes the $(k+1)$-provider. With this background, we now develop the formal pivot semantics for parameterized $\mathsf{TSO}(\mathsf{A})$.

Formal Pivot semantics for Parameterized $\mathsf{TSO}(\mathsf{A})$. We define the formal operational semantics of the pivot abstraction as a labelled transition system. Given a process description $\mathsf{Proc} = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$ and ADT $\mathsf{A} = \langle \mathsf{Vals}, \{\mathsf{val}_{\mathsf{init}}\}, \mathsf{Ops}, \longrightarrow_{\mathsf{A}} \rangle$, a configuration of the pivot transition system represents the *view* of a provider when simulating a run of the program. A view $\mathsf{v} = \langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle$ is defined as follows. The process state is given by $\mathsf{q} \in \mathsf{Q}$. The value of the provider's ADT A is $\mathsf{val} \in \mathsf{Vals}$. The function $\mathsf{Lw} : \mathsf{Vars} \longrightarrow \mathsf{Dom} \cup \{\oslash\}$ gives for each $\mathsf{x} \in \mathsf{Vars}$, the value of the latest (i.e., most recent) write the provider has performed on x. If no such instruction exists (the process has made no writes to x) then $\mathsf{Lw}(\mathsf{x}) = \oslash$. Note that Lw abstracts the buffer in terms of read-own-write operations since the process can only read from the most recent pending write in its buffer on each variable (if it exists). We define $\mathsf{Lw}_\oslash$ such that $\mathsf{Lw}_\oslash(\mathsf{x}) = \oslash$ for all $\mathsf{x} \in \mathsf{Vars}$. The first update sequence of pivot messages is $\omega \in \mathsf{Msgs}_{\mathsf{diff}}$. It is unchanged by transitions and remains constant throughout the pivot run.

The *external pointer*, $\phi_E \in \{0, 1, \ldots, |\omega|\}$ helps the provider keep track of which messages from ω it has observed. These messages have been propagated by other processes. The external pointer is used to identify which variables are still holding their initial values in the memory. If the provider observes an external write on a variable x (by accessing the memory), then this write has overwritten the initial value of x in the memory. The *local pointer* $\phi_L : \mathsf{Vars} \longrightarrow \{0, 1, \ldots, |\omega|\}$ is a set of pointers, one for each variable $\mathsf{x} \in \mathsf{Vars}$. The function $\phi_L(\mathsf{x})$ gives the highest ranked write operation the provider itself has performed (on any variable) before it performed the latest write on x. The local pointer is necessary to know which variables lose their initial values when we need to empty the buffer. In other words, the local pointer abstracts the buffer in terms of update operations. We define $\phi_L^{\max} := \max\{\phi_L(\mathsf{x}) \mid \mathsf{x} \in \mathsf{Vars}\}$ as the highest value of a local pointer and ϕ_L^0 such that $\phi_L^0(\mathsf{x}) = 0$ for all variables $\mathsf{x} \in \mathsf{Vars}$, i.e., the pointers are all in the leftmost position. The *progress pointer* $\phi_P \in \{1, 2, \ldots, |\omega| + 1\}$ gives the rank of the process the current provider is simulating.

$$\text{skip}\ \frac{\langle \mathsf{q}, \mathtt{skip}, \mathsf{q}' \rangle \in \delta}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{skip}}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle}$$

$$\text{write(1)}\ \frac{\langle \mathsf{q}, \mathtt{wr}(\mathsf{x}, \mathsf{d}), \mathsf{q}' \rangle \in \delta, \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle) < \phi_P, \phi'_L = \phi_L[\mathsf{x} \leftarrow \max(\mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle), \phi_L^{\max})]}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{wr}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}[\mathsf{x} \leftarrow \mathsf{d}], \omega, \phi_E, \phi'_L, \phi_P \rangle}$$

$$\text{write(2)}\ \frac{\langle \mathsf{q}, \mathtt{wrx}, \mathsf{d}, \mathsf{q}' \rangle \in \delta, \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle) = \phi_P}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{wr}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \mathsf{v}_{\mathsf{init}}(\omega, \phi_P + 1)}$$

$$\text{read(1)}\ \frac{\langle \mathsf{q}, \mathtt{rd}(\mathsf{x}, \mathsf{d}), \mathsf{q}' \rangle \in \delta, \mathsf{Lw}(\mathsf{x}) = \mathsf{d}}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{rd}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle}$$

$$\text{read(2)}\ \frac{\langle \mathsf{q}, \mathtt{rd}(\mathsf{x}, \mathsf{d}), \mathsf{q}' \rangle \in \delta, \mathsf{d} = \mathsf{init}(\mathsf{x}), \mathsf{Lw}(\mathsf{x}) = \bot, \mathrm{pos}(\omega)(\mathsf{x}) > \phi_E}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{rd}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle}$$

$$\text{read(3)}\ \frac{\langle \mathsf{q}, \mathtt{rd}(\mathsf{x}, \mathsf{d}), \mathsf{q}' \rangle \in \delta, \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle) < \phi_P, \phi'_E = \max(\phi_E, \phi_L(\mathsf{x}), \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle))}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{rd}(\mathsf{x},\mathsf{d})}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi'_E, \phi_L, \phi_P \rangle}$$

$$\text{memory-fence}\ \frac{\langle \mathsf{q}, \mathtt{mf}, \mathsf{q}' \rangle \in \delta, \phi'_E = \max(\phi_E, \phi_L^{\max})}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathtt{mf}}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}, \mathsf{Lw}, \omega, \phi'_E, \phi_L, \phi_P \rangle}$$

$$\text{data-operation}\ \frac{\langle \mathsf{q}, \mathsf{op}, \mathsf{q}' \rangle \in \delta, \mathsf{op} \in \mathsf{Ops}, \mathsf{val} \xrightarrow{\mathsf{op}}_{\mathsf{A}} \mathsf{val}'}{\langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle \xrightarrow{\mathsf{op}}_{\mathsf{pvt}} \langle \mathsf{q}', \mathsf{val}', \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle}$$

Fig. 1: The transition relation of the pivot semantics for a process Proc.

Given an update sequence $\omega \in \mathsf{Msgs}^{\mathsf{diff}}$ and $1 \leq k \leq |\omega| + 1$, we define *the initial view induced by* ω *and* k denoted by $\mathsf{v}_{\mathsf{init}}(\omega, k)$, as the view $\langle \mathsf{q}^{\mathsf{init}}, \mathsf{val}_{\mathsf{init}}, \mathsf{Lw}_\bot, \omega, 0, \phi_L^0, k \rangle$. For a given ω, the k-provider starts with $\mathsf{v}_{\mathsf{init}}(\omega, k)$: $\mathsf{Lw}_\bot$ and ϕ_L^0 imply that the simulated process has not performed any writes and $\phi_E = 0$ means that it has not read/updated from/to the memory.

We define the labeled transition relation $\longrightarrow_{\mathsf{pvt}}$ on the set of views by the inference rules given in Figure 1. The set of labels is $\mathsf{Instrs} \cup \mathsf{Ops}$. We describe the inference rules briefly. The `skip` rule only changes the local state of the process. There are two inference rules, write(1) and write(2), to describe the execution of a write operation $\mathtt{wr}(\mathsf{x}, \mathsf{d})$. The rule write(1) describes the situation when the rank of $\langle \mathsf{x}, \mathsf{d} \rangle$ is strictly smaller than the progress pointer ϕ_P. In this case, we update both Lw and ϕ_L. The rule write(2) describes the situation when the rank of $\langle \mathsf{x}, \mathsf{d} \rangle$ equals the progress pointer. This means that the provider has provided the message $\langle \mathsf{x}, \mathsf{d} \rangle$ with rank ϕ_P. Hence it has completed its mission, and initiates the next provider by transitioning to $\mathsf{v}_{\mathsf{init}}(\omega, \phi_P + 1)$.

There are three inference rules that describe a read operation $\mathtt{rd}(\mathsf{x}, \mathsf{d})$. The rule read(1) describes when the last written value to x by the provider is d, $\mathsf{Lw}(\mathsf{x}) = \mathsf{d}$. In this case, the provider simply reads from its local buffer. The rule read(2) describes the read of an initial value. It ensures that the read is possible by checking that no write operation on x is executed by the provider ($\mathsf{Lw}(\mathsf{x}) = \bot$), and by checking that the initial value of the variable has not been overwritten in the memory. This is achieved by checking if the position of $\langle \mathsf{x}, \mathsf{d} \rangle$ in ω, i.e. $\mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle)$, is strictly larger than ϕ_E. The rule read(3) describes when the simulated process reads from the memory. It checks that the message $\langle \mathsf{x}, \mathsf{d} \rangle$ has been generated by some previous provider ($\mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle) < \phi_P$), and then it updates the external pointer to $\max(\phi_E, \phi_L(\mathsf{x}), \mathrm{pos}(\omega)(\langle \mathsf{x}, \mathsf{d} \rangle))$. The memory fence rule describes when the simulated process does a fence action. The rule updates the external pointer to $\max(\phi_E, \phi_L^{\max})$. Finally, the data-operation rule describes when the simulated process does an ADT operation.

The set of initial views is $\mathsf{V}_{\mathsf{init}} = \{\mathsf{v}_{\mathsf{init}}(\omega, 1) \mid \omega \in \mathsf{Msgs}^{\mathsf{diff}}\}$. This is the set of initial views of the 1-provider and it is finite because $\mathsf{Msgs}^{\mathsf{diff}}$ is finite, unlike the set of initial configurations $\mathsf{C}_{\mathsf{init}}$ in the parameterized case under TSO.

6 Register Machines

Our goal is to design a general method to determine the decidability and complexity of $\mathsf{TSO(A)}$-`P-Reach` depending on A. We examine the pivot abstraction introduced in the previous chapter. A view $\mathsf{v} = \langle \mathsf{q}, \mathsf{val}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P \rangle$ of the pivot transition system, can be partitioned into the following two components: (1) $\mathsf{q}, \mathsf{Lw}, \omega, \phi_E, \phi_L, \phi_P$ which contains the local state and also effectively abstracts the unbounded FIFO buffers and shared memory of the TSO system and (2) val which captures the value of the ADT. The first part is finite since each component takes finitely many values. We call this the *book-keeping* state since it keeps track of the progress of the core TSO system. However, the ADT part can be infinite, depending upon the abstract data type.

We will use a register machine in order to represent the book-keeping state in a finite way using states and registers. On the other hand, we will keep the ADT component general and only later instantiate it to some interesting cases.

A *register machine* is a finite state automaton that has access to a finite set of *registers*, each holding a natural number. The register machine can execute two

operations on a register, it can write a given value or it can read a given value. A read is blocking if the given value is not in the register. We differ from most definitions of register machines in two significant ways: Since we only require a finite domain to model $\mathsf{TSO}(\mathsf{A})$ semantics, the values of the registers are bound from above by an $N \in \mathbb{N}$. This makes the register assignments finite whereas most definitions allow for an unbounded domain. Further, our register machine is augmented with an ADT.

Given an ADT $\mathsf{A} = \langle \mathsf{Vals}, \{\mathsf{val}_{\mathsf{init}}\}, \mathsf{Ops}, \longrightarrow_{\mathsf{A}} \rangle$, let Regs be a finite set of registers and $\mathsf{Dom} = \{0, \ldots, N\}$ their domain. We define the set of actions $\mathsf{Acts} = \{\mathtt{SKP}, \mathtt{WRITE}(\mathsf{r}, \mathsf{d}), \mathtt{READ}(\mathsf{r}, \mathsf{d}) \mid \mathsf{r} \in \mathsf{Regs}, \mathsf{d} \in \mathsf{Dom}\}$. A register machine is then defined as a tuple $\mathcal{R}(\mathsf{A}) = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$, where Q is a finite set of states, $\mathsf{q}_{\mathsf{init}} \in \mathsf{Q}$ is the initial state and $\delta \subseteq \mathsf{Q} \times (\mathsf{Acts} \cup \mathsf{Ops}) \times \mathsf{Q}$ is the transition relation.

The semantics of the register machine are given in terms of a transition system. The set of configurations is $\mathsf{Q} \times \mathsf{Dom}^{\mathsf{Regs}} \times \mathsf{Vals}$. A configuration consists of a state, a register assignment $\mathsf{Regs} \longrightarrow \mathsf{Dom}$ and a value of A. The initial configuration is $\langle \mathsf{q}_{\mathsf{init}}, 0^{\mathsf{Regs}}, \mathsf{val}_{\mathsf{init}} \rangle$, where all registers contain the value 0.

The transition relation $\longrightarrow$ is described in the following. $\mathtt{SKP}$ only changes the local state, not the registers or the ADT value. $\mathtt{WRITE}(\mathsf{r}, \mathsf{d})$ sets the value of the register r to d. $\mathtt{READ}(\mathsf{r}, \mathsf{d})$ is only enabled if the value of r is d, it does not change the value. The operations in Ops work as usually, they do not change any register. We define the state reachability problem for register machines as $\mathcal{R}(\mathsf{A})$-$\mathtt{Reach}$ in the usual way. A state $\mathsf{q}_{\mathsf{final}} \in \mathsf{Q}$ is reachable if there is a run of the transition system defined by the semantics of $\mathcal{R}(\mathsf{A})$ that starts in the initial configuration and ends in a configuration with state $\mathsf{q}_{\mathsf{final}}$.

6.1 Simulating Pivot Abstraction by Register Machines

In this section we will show how to simulate the pivot abstraction by a register machine. The idea is to save the book-keeping state (except for the local state) in the registers. Given a process description $\mathsf{Proc} = \langle \mathsf{Q}^{\mathsf{Proc}}, \mathsf{q}_{\mathsf{init}}^{\mathsf{Proc}}, \delta^{\mathsf{Proc}} \rangle$ for an ADT A, we construct a register machine $\mathcal{R}(\mathsf{A}) = \langle \mathsf{Q}, \mathsf{q}_{\mathsf{init}}, \delta \rangle$ that simulates the pivot semantics as follows. The set of registers is

$$\mathsf{Regs} := \{\mathsf{Lw}(\mathsf{x}), \mathsf{rk}_{\mathsf{Vars}}(\mathsf{x}), \mathsf{rk}_{\mathsf{Msgs}}(\mathsf{msg}), \phi_E, \phi_L(\mathsf{x}), \phi_L^{\max}, \phi_P, \mathsf{rk}_{\mathsf{nxt}} \mid \mathsf{x} \in \mathsf{Vars}, \mathsf{msg} \in \mathsf{Msgs}\} \ .$$

The registers $\mathsf{rk}_{\mathsf{Vars}}(\mathsf{x})$ and $\mathsf{rk}_{\mathsf{Msgs}}(\mathsf{msg})$ hold the rank of each variable and message, respectively. This implicitly gives rise to an update sequence. The auxiliary register $\mathsf{rk}_{\mathsf{nxt}}$ is used to initialize the other rank registers, as will be explained later on. The remaining registers correspond to their respective counterparts in the pivot abstraction. Note that the number of registers is linear in the number of messages $|\mathsf{Msgs}|$. The domain of the registers is defined to be $\mathsf{Dom} = \{0, \ldots, |\mathsf{Msgs}| + 1\}$. Since the TSO memory domain is finite, we can assume w.l.o.g. that the memory values are positive integers. If $\mathsf{Lw}(\mathsf{x}) = 0$, it means that there has been no write on x and it still holds the initial value. The set of states Q contains $\mathsf{Q}^{\mathsf{Proc}} \cup \{\mathsf{q}_{\mathsf{init}}^{\mathcal{R}}(\mathsf{A}), \mathsf{q}_{\mathsf{init}}^{\mathsf{ptr}}\}$ as well as a number of (unnamed) auxiliary states that will be used in the following.

To simplify our construction, we will use additional operations on registers, instead of just WRITE and READ. We introduce different blocking comparisons between registers and values such as $==, <, \leq, \neq$, register assignments such as $\mathsf{r} := \mathsf{r}'$, and increments by one denoted as $\mathsf{r}{+}{+}$. A more detailed description of these instructions is given in [3].

The Initializer. The pivot semantics define an exponential number of initial states: one per possible update sequence. The register machine instead guesses an update sequence at the start of the execution and stores it in the rank registers. This part of the register machine is the *rank initializer* (shown in Figure 2 (a)). It uses the auxiliary register $\mathsf{rk}_{\mathsf{nxt}}$ to keep track of the next rank that is to be assigned. In a nondeterministic manner, the rank initializer chooses a so far unranked message and then it assigns the next rank to this message. If the variable of the message has no rank assigned yet, it updates the rank of the variable. Then it increases the $\mathsf{rk}_{\mathsf{nxt}}$ register and continues. After each rank assignment, the initializer can choose to stop the rank assignment. In that case, it initializes the register ϕ_P to 1 and finishes in the initial state of Proc.

In addition to the rank initializer, we have the *pointer initializer*. It is responsible for resetting all pointers except the process pointer to zero. The process pointer is incremented by one instead. This initializer is not executed in the beginning of the simulation, but between epochs of the pivot abstraction.

The simulator. The main part of this construction handles the simulation of the pivot abstraction. It contains $\mathsf{Q}^{\mathsf{Proc}}$ as well as several auxiliary states that are described in the following. It simulates each instruction of $\mathsf{TSO}(\mathsf{A})$. The skip instruction and the data instructions are carried out unchanged. A visualization of the remaining instructions is depicted in Figure 2. In case of a write instruction $\mathtt{wr}(\mathsf{x}, \mathsf{d})$, we first compare the rank of the write message with the process pointer. If they are equal, it means that the epoch is finished and the next process should start, therefore we jump to the first state of the pointer initializer. Otherwise, we set the last write pointer $\mathsf{Lw}(\mathsf{x})$ to d. Now, we ensure that ϕ_L^{max} is at least as large as the rank of $\langle \mathsf{x}, \mathsf{d} \rangle$ and finally we update the local pointer $\phi_L(\mathsf{x})$ to be equal to ϕ_L^{max}. For the memory fence instruction, it only needs to be ensured that the external pointer is at least as large as the maximum local pointer ϕ_L^{max}. For a read instruction $\mathtt{rd}(\mathsf{x}, \mathsf{d})$, if the last write to x was of value d, we can execute the read directly. Otherwise, after checking that the write can be performed by the current provider, we ensure that the external pointer is at least as large as both the rank of $\langle \mathsf{x}, \mathsf{d} \rangle$ and the local pointer of x. For the special case that $\mathsf{d} = \mathsf{d}_{\mathsf{init}}$, there is an additional way in which the read can be performed: We can read $\mathsf{d}_{\mathsf{init}}$ from the memory if the process has neither already written to x nor observed a write that has higher or equal rank than the rank of x. This gives us the following theorem, proven in Appendix C of the full version [3]:

Theorem 1. $\mathsf{TSO}(\mathsf{A})$*-P-Reach is polynomial time reducible to* $\mathcal{R}(\mathsf{A})$*-Reach.*

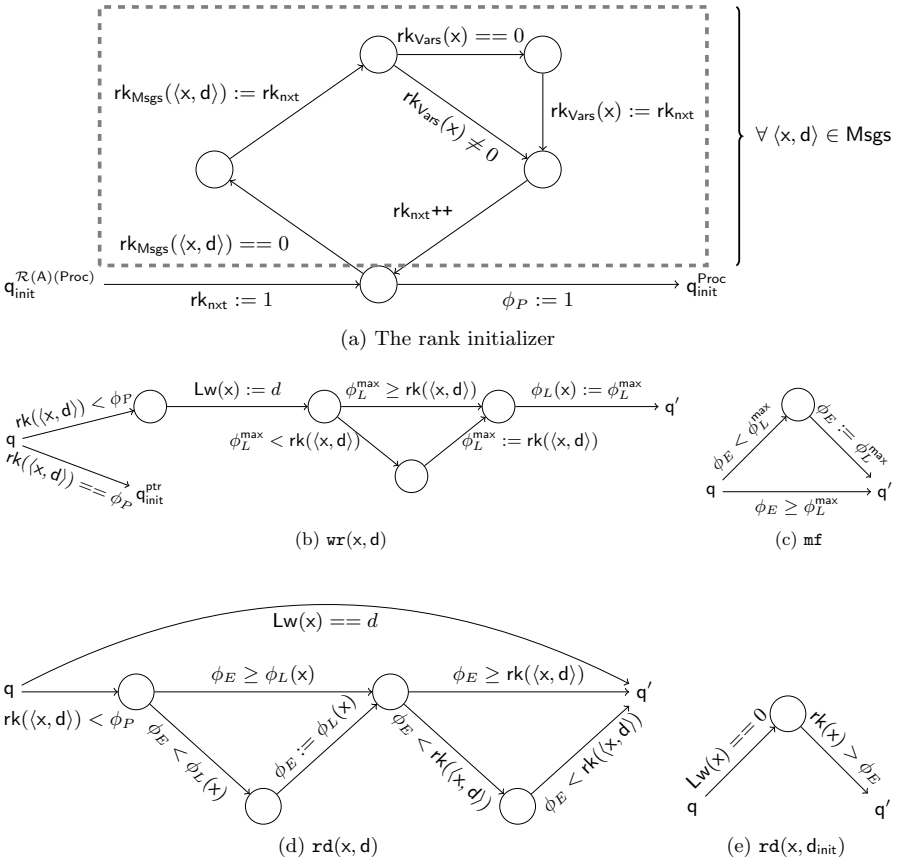


Fig. 2: The rank initializer and the simulator for some instructions instr.

6.2 Simulating Register Machines by TSO

We will now show how to simulate an ADT register machine with a parameterized program running under TSO(A). The main idea is to save the information about the registers in the last pending write operations, while making sure that not a single write operation actually hits the memory. Thus, the simulator always reads the initial value or its own writes, never writes of other processes.

The TSO program has a variable for each register, and two additional variables x_s and x_c that act as flags: x_s indicates that the verifier should start working, while x_c indicates that the verifier has successfully completed the verification. At the beginning of the execution, each process nondeterministically chooses to be either *simulator*, *scheduler*, or *verifier*. Each role will be described in the following. The complete construction is shown in Appendix C of [3].

The simulator uses the same states and transitions as $\mathcal{R}(\mathsf{A})$, but instead of reading from and writing to registers, it uses the memory. If the simulator reaches the target state $\mathsf{q}_{\mathsf{target}}$, it first checks the x_s flag. If it is already set, the simulator

stops, never reaching the final state $\mathsf{q}_{\mathsf{final}}$. Otherwise, it waits until it observes the flag x_c to be set. It then enters the final state. The scheduler's only responsibility is to signal the start of the verification process. It does so by setting the flag x_s at a nondeterministically chosen time during the execution of the program. The verifier waits until it observers the flag x_s. It then starts the verification process, which consists of checking each variable that corresponds to a register. If all of them still contain their initial value, the verification was successful. The verifier signals this to the simulator process by setting the x_c flag.

Any execution ending in $\mathsf{q}_{\mathsf{final}}$ must perform a simulation of $\mathcal{R}(\mathsf{A})$ ending in $\mathsf{q}_{\mathsf{target}}$ first, then a scheduler propagates the setting of flag x_s and afterwards a verifier executes. This ensures that the initial values are read by the verifier after the register machine has been simulated and thus the shared memory is unchanged. This means the simulator only accessed its write buffer and not writes from other threads. It follows that $\mathsf{q}_{\mathsf{target}}$ is reachable by $\mathcal{R}(\mathsf{A})$ if and only if $\mathsf{q}_{\mathsf{final}}$ is reachable by Proc under $\mathsf{TSO}(\mathsf{A})$. This gives us the following result:

Theorem 2. *$\mathcal{R}(\mathsf{A})$-Reach is polynomial time reducible to $\mathsf{TSO}(\mathsf{A})$-P-Reach.*

Theorem 1 and Theorem 2 give us a method of determining upper and lower bounds of the complexity of $\mathsf{TSO}(\mathsf{A})$-P-Reach for different instantiations of ADT. Since we have reductions in both directions, we can conclude that $\mathsf{TSO}(\mathsf{A})$-P-Reach is decidable if and only if $\mathcal{R}(\mathsf{A})$-Reach is decidable. We know $\mathsf{TSO}(\mathsf{A})$-P-Reach is PSPACE-hard for $\mathsf{TSO}(\mathtt{NoAdt})$-P-Reach where NoAdt is the trivial ADT that models plain TSO semantics [4]. We can immediately derive a lower bound for any ADT: $\mathsf{TSO}(\mathsf{A})$-P-Reach is PSPACE-hard.

7 Instantiations of ADTs

In the following, we instantiate our framework to a number of ADTs in order to show its applicability.

Theorem 3. *$\mathsf{TSO}(\mathtt{Ct})$-P-Reach and $\mathsf{TSO}(\mathtt{wCt})$-P-Reach are* PSPACE*-complete.*

We know $\mathsf{TSO}(\mathsf{A})$-P-Reach is PSPACE-hard for any ADT A including Ct and wCt. Regarding the upper bound for Ct, we can show that $\mathcal{R}(\mathtt{Ct})$-Reach can be polynomially reduced to $\mathcal{R}(\mathtt{NoAdt})$-Reach. The idea is to show that there is a bound on the counter values in order to find a witness for $\mathcal{R}(\mathtt{Ct})$-Reach. This bound is polynomial in the number of possible states and register assignments (i.e., this bound is at most exponential in the size of $\mathcal{R}(\mathrm{Ct})$.) Assume a run that contains a configuration c with a value that exceeds the bound, then certain state and register assignment are repeated in the run with different values. We can use this to shorten the run such that the counter value in c is reduced.

We can encode the counter value (up to this bound) in a binary way into registers acting as bits. The number of additional registers is polynomial in the size of $\mathcal{R}(\mathrm{Ct})$. In order to simulate an `inc` operation on this binary encoding using `WRITE` and `READ`, we only have to go through the bits starting at the least

important bit and flip them until one is flipped from 0 to 1. The dec operation works analogously. This only requires a polynomial state and transition overhead.

We know that $\mathcal{R}$(NoAdt)-Reach is in PSpace[4]. It follows from the polynomial reduction that $\mathcal{R}$(Ct)-Reach is in PSpace. Applying Theorem 1 gives us that TSO(Ct)-P-Reach is in PSpace. Since any wCt is a Ct, it follows TSO(wCt)-P-Reach is in PSpace as well. The proof is in [3].

Theorem 4. TSO(St)-*P-Reach is* ExpTime-*complete.*

For membership, we encode the registers of $\mathcal{R}$(St) in the states, which yields a finite state machine with access to a stack, i.e. a pushdown automaton. The construction has an exponential number of states. From [45], we have that checking the emptiness of a context-free language generated by a pushdown automaton is polynomial in terms of the size of the automaton. Combined, we get that state reachability of the constructed pushdown automaton is in ExpTime. It follows that $\mathcal{R}$(St)-Reach is in ExpTime (thanks to Theorem 1).

To prove the lower bound, we can reduce the problem of checking the emptiness of the intersection of a pushdown with n finite-state automata [35] to $\mathcal{R}$(St)-Reach. This problem is well-known to be ExpTime-complete. The idea is to use the stack to simulate pushdown automaton and n registers to keep track of the states of the finite-state automata. We apply Theorem 2 and get TSO(St)-P-Reach is ExpTime-hard. The formal proof is in [3]

Theorem 5. TSO(Petri)-*P-Reach is* ExpSpace-*complete.*

Proof. Petri net coverability is known to be ExpSpace complete [26]. We show hardness by reducing coverability of a marking m to $\mathcal{R}$(Petri)-Reach. The idea is to construct a register machine with a Petri net as ADT. This register machine will have two states $\mathsf{q}_{\mathsf{init}}$ and $\mathsf{q}_{\mathsf{final}}$. For every transition t of the original Petri net, we have t: $\mathsf{q}_{\mathsf{init}} \xrightarrow{t} \mathsf{q}_{\mathsf{init}}$ as a transition of the register machine (we simply simulate the original Petri net). Furthermore, we have $\mathsf{q}_{\mathsf{init}} \xrightarrow{t-m} \mathsf{q}_{\mathsf{final}}$ as a transition of the register machine. Thus, the state $\mathsf{q}_{\mathsf{final}}$ can be reached iff m can be covered.

We reduce reachability of $\mathcal{R}$(Petri) to Petri net coverability. We construct the Petri net by taking the ADT Petri and adding a place p_{q} for every state q and a place $p_{\mathsf{reg},d}$ for every register $\mathsf{reg} \in \mathsf{Regs}$ and register value $d \in \mathsf{Dom}$. The idea is that a marking with a token in p_{q} and one in $p_{\mathsf{reg},d}$ but none $p_{\mathsf{reg},d'}$ for $d' \neq d$ corresponds to a configuration of $\mathcal{R}$(Petri) with state q and $\mathsf{reg} = d$. The value of Petri is given by the remainder of the marking.

We simulate any $\mathsf{q} \xrightarrow{\mathsf{instr}} \mathsf{q}'$ with a transition t that takes one token from q and puts one in q'. If $\mathsf{instr} \in \mathsf{Ops}$, then instr is a Petri net transition. We simply add the same input and output arcs to t. To simulate a write, we add a new transition $t_{d'}$ for every $d' \in \mathsf{Dom}$ with an arc to $p_{\mathsf{reg},d}$ and an arc from $p_{\mathsf{reg},d'}$. The initial marking is consistent with $\mathsf{val}_{\mathsf{init}}^{\text{Petri}}$ and has one token in $p_{\mathsf{q}_{\mathsf{init}}}$. A state q is reachable if a marking with one token in p_{q} is coverable.

Higher Order ADTs. Let $\mathcal{M}(\mathsf{A})$-Reach problem be the restriction of $\mathcal{R}(\mathsf{A})$-Reach with no registers. The $\mathcal{M}(\mathsf{A})$-Reach problem has been studied for many ADT such as higher order counter and higher order stack variations[34,25].

Theorem 6.

- $\mathsf{TSO}(n\text{-}\mathtt{St})$-*P-Reach is* $(n-1)$-ExpTime-*hard and in* n-ExpTime.
- $\mathsf{TSO}(n\text{-}\mathtt{wCt})$-*P-Reach is* $(n-2)$-ExpTime-*hard and in* $(n-1)$-ExpTime.
- $\mathsf{TSO}(n\text{-}\mathtt{Ct})$-*P-Reach is* $(n-2)$-ExpSpace-*hard and in* $(n-1)$-ExpSpace.

Proof. $\mathcal{M}(n\text{-}\mathtt{St})\text{-}\mathtt{Reach}$ has been shown to be $(n-1)$-ExpTime-complete [25]. We know $\mathcal{M}(n\text{-}\mathtt{wCt})\text{-}\mathtt{Reach}$ is $(n-2)$-ExpTime-complete and $\mathcal{M}(n\text{-}\mathtt{Ct})\text{-}\mathtt{Reach}$ is $(n-2)$-ExpSpace-complete [34]. Since the reduction from $\mathcal{M}(\mathsf{A})\text{-}\mathtt{Reach}$ to $\mathcal{R}(\mathsf{A})\text{-}\mathtt{Reach}$ is trivial, any hardness result can be applied to $\mathsf{TSO}(\mathsf{A})\text{-}\mathtt{P}\text{-}\mathtt{Reach}$ immediately using Theorem 2. In order to reduce $\mathcal{R}(\mathsf{A})\text{-}\mathtt{Reach}$ to $\mathcal{M}(\mathsf{A})\text{-}\mathtt{Reach}$, we encode register assignments into the state which results in an exponential state explosion. Then we apply Theorem 1 to obtain our upper bound.

Theorem 7. $\mathsf{TSO}(n\text{-}\mathtt{OMSt})$-*P-Reach is* 2-ETime-*complete.*

Proof. We know that $\mathcal{M}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{Reach}$ is 2-ETime-complete [12] and we can apply Theorem 2 to get 2-ETime-hardness. According to Theroem 4.6 in [11], $\mathcal{M}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{Reach}$ is in $\mathcal{O}(|\mathcal{M}(\mathsf{A})|^{2^{dn}})$ for some constant $d \in \mathbb{N}$. We apply the exponential size reduction to $\mathcal{R}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{Reach}$ and Theorem 1 and get $\mathsf{TSO}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{P}\text{-}\mathtt{Reach}$ is in $\mathcal{O}((2^{|\mathcal{P}|})^{2^{dn}}) = \mathcal{O}(2^{|\mathcal{P}|\cdot 2^{dn}})$ and thus it is also in $\mathcal{O}(2^{2^{|\mathcal{P}|}\cdot 2^{dn}}) = \mathcal{O}(2^{2^{|\mathcal{P}|+dn}})$. Thus, $\mathsf{TSO}(n\text{-}\mathtt{OMSt})\text{-}\mathtt{P}\text{-}\mathtt{Reach}$ is in 2-ETime.

We study well structured ADTs [29,9] as defined in [3]:

Theorem 8. *If ADT* A *is well structured, then* $\mathsf{TSO}(\mathsf{A})$-*P-Reach is decidable.*

A register machine for a well structured ADT A is equivalent to the composition of a well structured transition system (WSTS) modeling A and a finite transition system (and thus a WSTS) that models states and registers. According to [9], the composition is again a WSTS and reachability is decidable. The above theorem is then an immediate corollary of Theorem 1.

8 Conclusions and Future Work

In this paper, we have taken the first step to studying the complexity of parameterized verification under weak memory models when the processes manipulate unbounded data domains. Concretely, we have presented complexity results for parameterized concurrent programs running on the classical TSO memory model when the processes operate on an abstract data type. We reduce the problem to reachability for register machines enriched with the given abstract data type.

State reachability for finite automata with ADT has been extensively studied for many ADTs[34,25]. We have shown in Theorem 6 that we can apply

our framework to existing complexity results of this problem. This provides us with decidability and complexity results for the corresponding instances of TSO(A)-P-Reach. However, due to the exponential number of register assignments, the upper bound is exponentially larger than the lower bound. We aim to study these cases further and determine more refined parametric bounds.

A direction for future work is considering other memory models, such as the partial store ordering semantics, the release-acquire semantics, and the ARM semantics. It is also interesting to re-consider the problem under the assumption of having distinguished processes (so-called *leader processes*). Adding leaders is known to make the parameterized verification problem harder. The complexity/decidability of parameterized verification under TSO with a single leader is open, even when the processes are finite-state.

References

1. Parosh Aziz Abdulla. Regular model checking. *STTT*, 14(2):109–118, 2012.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. A load-buffer semantics for total store ordering. *LMCS*, 14(1), 2018.
3. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Florian Furbach, Adwait Godbole, Yacoub G. Hendi, Shankaranarayanan Krishna, and Stephan Spengler. Parameterized verification under tso with data types. *arXiv e-prints*, 2023. arXiv:2302.02163.
4. Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Rojin Rezvan. Parameterized verification under tso is pspace-complete. *Proc. ACM Program. Lang.*, 4(POPL), 2019.
5. Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine. Constrained monotonic abstraction: A CEGAR for parameterized verification. In *CONCUR 2010*, pages 86–101, 2010.
6. Parosh Aziz Abdulla and Giorgio Delzanno. Parameterized verification. *STTT*, 18(5):469–473, 2016.
7. Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. Parameterized verification through view abstraction. *STTT*, 18(5):495–516, 2016.
8. Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. In *Handbook of Model Checking*, pages 685–725. Springer, 2018.
9. Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160:109–127, 2000.
10. Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.
11. Mohamed Faouzi Atig. Model-Checking of Ordered Multi-Pushdown Automata. *LMCS*, Volume 8, Issue 3, 2012.
12. Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In *Developments in Language Theory*, pages 121–133. Springer, 2008.
13. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability in parameterized verification. *SIGACT News*, 47(2):53–64, 2016.

14. Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large (extended abstract). In *CAV*, volume 2725 of *LNCS*, pages 223–235. Springer, 2003.
15. Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ETAPS*, pages 533–553, 2013.
16. Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular (tree) model checking. *STTT*, 14(2):167–191, 2012.
17. Sebastian Burckhardt. Principles of eventual consistency. *FTPL*, 1(1-2):1–150, 2014.
18. Thierry Cachat and Igor Walukiewicz. The complexity of games on higher order pushdown automata. *CoRR*, abs/0705.0262, 2007.
19. Sylvain Conchon, David Declerck, and Fatiha Zaïdi. Parameterized model checking on the tso weak memory model. *J. Autom. Reason.*, 64(7):1307–1330, 2020.
20. Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. Parameterized verification of ad hoc networks. In *CONCUR*, pages 313–327, 2010.
21. Marco Elver and Vijay Nagarajan. TSO-CC: consistency directed cache coherence for TSO. In *HPCA*, pages 165–176. IEEE, 2014.
22. E. Allen Emerson, John Havlicek, and Richard J. Trefler. Virtual symmetry reduction. In *LICS*, pages 121–131, 2000.
23. E. Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *CHARME*, volume 2860 of *LNCS*, pages 247–262. Springer, 2003.
24. E. Allen Emerson and Vineet Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL*, volume 3210 of *LNCS*, pages 325–339. Springer, 2004.
25. Joost Engelfriet. Iterated stack automata and complexity classes. *Inf. Comput.*, 95(1):21–75, 1991.
26. Javier Esparza. Decidability and complexity of petri net problems - an introduction. *LNCS*, 1491, 2000.
27. Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999.
28. Javier Esparza, Pierre Ganty, and Rupak Majumdar. Parameterized verification of asynchronous shared-memory systems. *J. ACM*, 63(1):10:1–10:48, 2016.
29. A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001. ISS.
30. Marie Fortin, Anca Muscholl, and Igor Walukiewicz. Model-checking linear-time properties of parametrized asynchronous shared-memory pushdown systems. In *CAV*, pages 155–175, 2017.
31. Pierre Ganty and Rupak Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6:1–6:48, 2012.
32. Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.
33. Matthew Hague. Parameterised pushdown systems with non-atomic writes. In *FSTTCS*, pages 457–468, 2011.
34. Alexander Heußner and Alexander Kartzow. Reachability in higher-order-counters. In *MFCS*, pages 528–539. Springer, 2013.
35. Alexander Heußner, Jérôme Leroux, Anca Muscholl, and Grégoire Sutre. Reachability analysis of communicating pushdown systems. In *FOSSACS*, pages 267–281. Springer, 2010.
36. Vineet Kahlon. Parameterization as abstraction: A tractable approach to the dataflow analysis of concurrent programs. In *LICS*, pages 181–192, 2008.

37. Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, volume 6174 of *LNCS*, pages 645–659. Springer, 2010.
38. Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.*, 256(1-2):93–112, 2001.
39. Shankara Narayanan Krishna, Adwait Godbole, Roland Meyer, and Soham Chakraborty. Parameterized verification under release acquire is pspace-complete. In *PODC*, pages 482–492. ACM, 2022.
40. Salvatore La Torre, Anca Muscholl, and Igor Walukiewicz. Safety of parametrized asynchronous shared-memory systems is almost always decidable. In *CONCUR*, pages 72–84, 2015.
41. Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *SIGPLAN-SIGACT*, pages 649–662. ACM, 2016.
42. Anca Muscholl, Helmut Seidl, and Igor Walukiewicz. Reachability for dynamic parametric processes. In *VMCAI*, pages 424–441, 2017.
43. Kedar S. Namjoshi and Richard J. Trefler. Parameterized compositional model checking. In *ETAPS*, volume 9636 of *LNCS*, pages 589–606. Springer, 2016.
44. J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981.
45. Ahmed Bouajjani Rajeev Alur and Javier Esparza. *Handbook of Model Checking*, chapter Model Checking Procedural Programs, pages 547–569. Springer, 2018.
46. Alberto Ros and Stefanos Kaxiras. Racer: TSO consistency via race detection. In *MICRO*, pages 33:1–33:13. IEEE Computer Society, 2016.
47. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *ACM SIGPLAN, PLDI*, pages 175–186. ACM, 2011.
48. Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

Verifying Learning-Based Robotic Navigation Systems

Guy Amir[1,*(✉)], Davide Corsi[2,*], Raz Yerushalmi[1,3], Luca Marzari[2], David Harel[3], Alessandro Farinelli[2], and Guy Katz[1]

[1] The Hebrew University of Jerusalem, Jerusalem, Israel
{guyam,guykatz}@cs.huji.ac.il
[2] University of Verona, Verona, Italy
{davide.corsi,luca.marzari,alessandro.farinelli}@univr.it
[3] The Weizmann Institute of Science, Rehovot, Israel
{raz.yerushalmi,david.harel}@weizmann.ac.il

Abstract. Deep reinforcement learning (DRL) has become a dominant deep-learning paradigm for tasks where complex policies are learned within reactive systems. Unfortunately, these policies are known to be susceptible to bugs. Despite significant progress in DNN verification, there has been little work demonstrating the use of modern verification tools on real-world, DRL-controlled systems. In this case study, we attempt to begin bridging this gap, and focus on the important task of mapless robotic navigation — a classic robotics problem, in which a robot, usually controlled by a DRL agent, needs to efficiently and safely navigate through an unknown arena towards a target. We demonstrate how modern verification engines can be used for effective *model selection*, i.e., selecting the best available policy for the robot in question from a pool of candidate policies. Specifically, we use verification to detect and rule out policies that may demonstrate suboptimal behavior, such as collisions and infinite loops. We also apply verification to identify models with overly conservative behavior, thus allowing users to choose superior policies, which might be better at finding shorter paths to a target. To validate our work, we conducted extensive experiments on an actual robot, and confirmed that the suboptimal policies detected by our method were indeed flawed. We also demonstrate the superiority of our verification-driven approach over state-of-the-art, gradient attacks. Our work is the first to establish the usefulness of DNN verification in identifying and filtering out suboptimal DRL policies in real-world robots, and we believe that the methods presented here are applicable to a wide range of systems that incorporate deep-learning-based agents.

1 Introduction

In recent years, *deep neural networks* (DNN) have become extremely popular, due to achieving state-of-the-art results in a variety of fields — such as natural

[*] Both authors contributed equally.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 607–627, 2023.
https://doi.org/10.1007/978-3-031-30823-9_31

language processing [16], image recognition [51], autonomous driving [11], and more. The immense success of these DNN models is owed in part to their ability to train on a fixed set of training samples drawn from some distribution, and then *generalize*, i.e., correctly handle inputs that they had not encountered previously. Notably, *deep reinforcement learning* (DRL) [37] has recently become a dominant paradigm for training DNNs that implement control policies for complex systems that operate within rich environments. One domain in which DRL controllers have been especially successful is robotics, and specifically — robotic navigation, i.e., the complex task of efficiently navigating a robot through an arena, in order to safely reach a target [63, 68].

Unfortunately, despite the immense success of DNNs, they have been shown to suffer from various safety issues [31, 57]. For example, small perturbations to their inputs, which are either intentional or the result of noise, may cause DNNs to react in unexpected ways [45]. These inherent weaknesses, and others, are observed in almost every kind of neural network, and indicate a need for techniques that can supply formal guarantees regarding the safety of the DNN in question. These weaknesses have also been observed in DRL systems [6,21,34], showing that even state-of-the-art DRL models may err miserably.

To mitigate such safety issues, the verification community has recently developed a plethora of techniques and tools [8,10,19,24,28,29,31,35,39,40,64,66] for formally verifying that a DNN model is safe to deploy. Given a DNN, these methods usually check whether the DNN: (i) behaves according to a prescribed requirement for *all* possible inputs of interest; or (ii) violates the requirement, in which case the verification tool also provides a counterexample.

To date, despite the abundance of both DRL systems and DNN verification techniques, little work has been published on demonstrating the applicability and usefulness of verification techniques to real-world DRL systems. In this case study, we showcase the capabilities of DNN verification tools for analyzing DRL-based systems in the robotics domain — specifically, robotic navigation systems. To the best of our knowledge, this is the first attempt to demonstrate how off-the-shelf verification engines can be used to identify both *unsafe* and *suboptimal* DRL robotic controllers, that cannot be detected otherwise using existing, incomplete methods. Our approach leverages existing DNN verifiers that can reason about single and multiple invocations of DRL controllers, and this allows us to conduct a verification-based model selection process — through which we filter out models that could render the system unsafe.

In addition to model selection, we demonstrate how verification methods allow gaining better insights into the DRL training process, by comparing the outcomes of different training methods and assessing how the models improve over additional training iterations. We also compare our approach to gradient-based methods, and demonstrate the advantages of verification-based tools in this setting. We regard this as another step towards increasing the reliability and safety of DRL systems, which is one of the key challenges in modern machine learning [27]; and also as a step toward a more wholesome integration of verification techniques into the DRL development cycle.

In order to validate our experiments, we conducted an extensive evaluation on a real-world, physical robot. Our results demonstrate that policies classified as suboptimal by our approach indeed exhibited unwanted behavior. This evaluation highlights the practical nature of our work; and is summarized in a short video clip [4], which we strongly encourage the reader to watch. In addition, our code and benchmarks are available online [3].

The rest of the paper is organized as follows. Section 2 contains background on DNNs, DRLs, and robotic controlling systems. In Section 3 we present our DRL robotic controller case study, and then elaborate on the various properties that we considered in Section 4. In Section 5 we present our experimental results, and use them to compare our approach with competing methods. Related work appears in Section 6, and we conclude in Section 7.

2 Background

Deep Neural Networks. Deep neural networks (DNNs) [25] are computational, directed, graphs consisting of multiple layers. By assigning values to the first layer of the graph and propagating them through the subsequent layers, the network computes either a label prediction (for a classification DNN) or a value (for a regression DNN), which is returned to the user. The values computed in each layer depend on values computed in previous layers, and also on the current layer's *type*. Common layer types include the *weighted sum* layer, in which each neuron is an affine transformation of the neurons from the preceding layer; as well as the popular *rectified linear unit* (*ReLU*) layer, where each node y computes the value $y = \text{ReLU}(x) = \max(0, x)$, based on a single node x from the preceding layer to which it is connected. The DRL systems that are the subject matter of this case study consist solely of weighted sum and ReLU layers, although the techniques mentioned are suitable for DNNs with additional layer types, as we discuss later.

Fig. 1 depicts a small example of a DNN. For input $V_1 = [2, 3]^T$, the second (weighted sum) layer computes the values $V_2 = [20, -7]^T$. In the third layer, the ReLU functions are applied, and the result is $V_3 = [20, 0]^T$. Finally, the network's single output is computed as a weighted sum: $V_4 = [40]$.

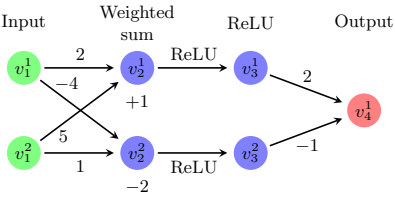


Fig. 1: A toy DNN.

Deep Reinforcement Learning. Deep reinforcement learning (DRL) [37] is a particular paradigm and setting for training DNNs. In DRL, an *agent* is trained to learn a *policy* π, which maps each possible *environment state* s (i.e., the current observation of the agent) to an *action* a. The policy can have different interpretations among various learning algorithms. For example, in some cases, π represents a probability distribution over the action space, while in others it encodes a function that estimates a *desirability score* over all the future actions from a state s.

During training, at each discrete time-step $t \in \{0, 1, 2, \ldots\}$, a *reward* r_t is presented to the agent, based on the action a_t it performed at time-step t. Different DRL training algorithms leverage the reward in different ways, in order to optimize the DNN-agent's parameters during training. The general DNN architecture described above also characterizes DRL-trained DNNs; the uniqueness of the DRL paradigm lies in the training process, which is aimed at generating a DNN that computes a mapping π that maximizes the *expected cumulative discounted reward* $R_t = \mathbb{E}\left[\sum_t \gamma^t \cdot r_t\right]$. The *discount factor*, $\gamma \in [0, 1]$, is a hyperparameter that controls the influence that past decisions have on the total expected reward.

DRL training algorithms are typically divided into three categories [55]:

1. **Value-Based Algorithms.** These algorithms attempt to learn a value function (called the *Q-function*) that assigns a value to each ⟨state,action⟩ pair. This iterative process relies on the *Bellman equation* [44] to update the function: $\mathbb{Q}^\pi(s_t, a_t) = r + \gamma \max_{a_{t+1}} \mathbb{Q}^\pi(s_{t+1}, a_{t+1})$. *Double Deep Q-Network* (DDQN) is an optimized implementation of this algorithm [60].
2. **Policy-Gradient Algorithms.** This class contains algorithms that attempt to directly learn the optimal policy, instead of assessing the value function. The algorithms in this class are typically based on the *policy gradient theorem* [56]. A common implementation is the *Reinforce* algorithm [67], which aims to directly optimize the following objective function, over the parameters θ of the DNN, through a gradient ascent process: $\nabla_\theta \mathbb{J}(\pi_\theta) = \mathbb{E}[\sum_t^T \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot r_t]$. For additional details, see [67].
3. **Actor-Critic Algorithms.** This family of hybrid algorithms combines the two previous approaches. The key idea is to use two different neural networks: a *critic*, which learns the value function from the data, and an *actor*, which iteratively improves the policy by maximizing the value function learned by the critic. A state-of-the-art implementation of this approach is the *Proximal Policy Optimization* (PPO) algorithm [50].

All of these approaches are commonly used in modern DRL; and each has its advantages and disadvantages. For example, the value-based methods typically require only small sets of examples to learn from, but are unable to learn policies for continuous spaces of ⟨state,action⟩ pairs. In contrast, the policy-gradient methods can learn continuous policies, but suffer from a low sample efficiency and large memory requirements. Actor-Critic algorithms attempt to combine the benefits of value-based and policy-gradient methods, but suffer from high instability, particularly in the early stages of training, when the value function learned by the critic is unreliable.

DNN Verification and DRL Verification. A DNN verification algorithm receives as input [31]: (i) a trained DNN N; (ii) a precondition P on the DNN's inputs, which limits their possible assignments to inputs of interest; and (iii) a postcondition Q on N's output, which usually encodes the *negation* of the behavior we would like N to exhibit on inputs that satisfy P. The verification algorithm then searches for a concrete input x_0 that satisfies $P(x_0) \wedge Q(N(x_0))$,

and returns one of the following outputs: (i) `SAT`, along with a concrete input x_0 that satisfies the given constraints; or (ii) `UNSAT`, indicating that no such x_0 exists. When Q encodes the negation of the required property, a `SAT` result indicates that the property is violated (and the returned input x_0 triggers a bug), while an `UNSAT` result indicates that the property holds.

For example, suppose we wish to verify that the DNN in Fig. 1 always outputs a value strictly smaller than 7; i.e., that for any input $x = \langle v_1^1, v_1^2 \rangle$, it holds that $N(x) = v_4^1 < 7$. This is encoded as a verification query by choosing a precondition that does not restrict the input, i.e., $P = (true)$, and by setting $Q = (v_4^1 \geq 7)$, which is the *negation* of our desired property. For this verification query, a sound verifier will return `SAT`, alongside a feasible counterexample such as $x = \langle 0, 2 \rangle$, which produces $v_4^1 = 22 \geq 7$. Hence, the property does not hold for this DNN.

To date, the DNN verification community has focused primarily on DNNs used for a single, non-reactive, invocation [24,28,31,40,64]. Some work has been carried out on verifying DRL networks, which pose greater challenges: beyond the general scalability challenges of DNN verification, in DRL verification we must also take into account that agents typically interact with a reactive environment [6,9,15,21,30]. In particular, these agents are implemented with neural networks that are invoked multiple times, and the inputs of each invocation are usually affected by the outputs of the previous invocations. This fact aggregates the scalability limitations (because multiple invocations must be encoded in each query), and also makes the task of defining P and Q significantly more complex [6].

3 Case Study: Robotic Mapless Navigation

Robotis Turtlebot 3. In our case study, we focus on the *Robotis Turtlebot 3* robot (*Turtlebot*, for short), depicted in Fig. 2. Given its relatively low cost and efficient sensor configuration, this robot is widely used in robotics research [7,46]. In particular, this robotic platform has the actuators required for moving and turning, as well as multiple lidar sensors for detecting obstacles. These sensors use laser beams to approximate the distance to the nearest object in their direction [65]. In our experiments, we used a configuration with seven lidar sensors, each with a maximal range of one meter. Each pair of sensors are 30° apart, thus allowing coverage of 180°. The images in Fig. 3 depict a simulation of the Turtlebot navigating through an arena, and highlight the lidar beams. See the full version of this paper [5] for additional details.

The Mapless Navigation Problem. *Robotic navigation* is the task of navigating a robot (in our case, the Turtlebot) through an arena. The robot's goal is to reach a target destination while adhering to predefined restrictions; e.g., selecting as short a path as possible, avoiding obstacles, or optimizing energy consumption. In recent years, robotic navigation tasks have received a great deal of attention [63,68], primarily due to their applicability to autonomous vehicles.

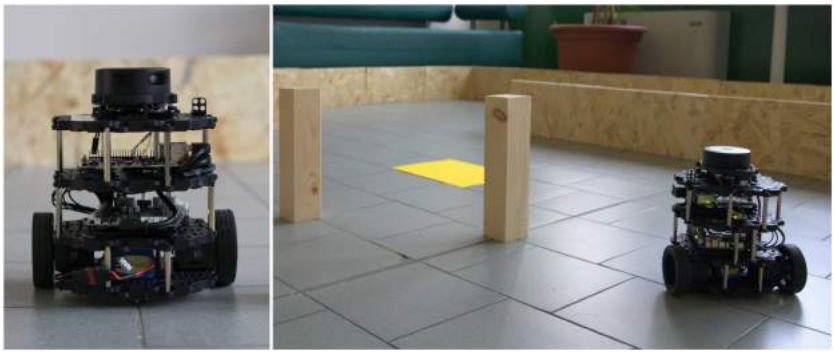


Fig. 2: The *Robotis Turtlebot 3* platform, navigating in an arena. The image on the left depicts a static robot, and the image on the right depicts the robot moving towards the destination (the yellow square), while avoiding two wooden obstacles in its route.

We study here the popular *mapless* variant of the robotic navigation problem, where the robot can rely only on local observations (i.e., its sensors), without any information about the arena's structure or additional data from external sources. In this setting, which has been studied extensively [58], the robot has access to the *relative location* of the target, but does not have a *complete map* of the arena. This makes mapless navigation a partially observable problem, and among the most challenging tasks to solve in the robotics domain [13, 58, 70].

DRL-Controlled Mapless Navigation. State-of-the-art solutions to mapless navigation suggest training a DRL policy to control the robot. Such DRL-based solutions have obtained outstanding results from a performance point of view [47]. For example, recent work by Marchesini et al. [43] has demonstrated how DRL-based agents can be applied to control the Turtlebot in a mapless navigation setting, by training a DNN with a simple architecture, including two hidden layers. Following this recent work, in our case study we used the following topology for DRL policies:

- An input layer with nine neurons. These include seven neurons representing the Turtlebot's lidar readings. The additional, non-lidar inputs include one neuron representing the relative angle between the robot and the target, and one neuron representing the robot's distance from the target. A scheme of the inputs appears in Fig. 4a.
- Two subsequent fully-connected layers, each consisting of 16 neurons, and followed by a ReLU activation layer.
- An output layer with three neurons, each corresponding to a different (discrete) action that the agent can choose to execute in the following step: move `FORWARD`, turn `LEFT`, or turn `RIGHT`.[1]

[1] It has been shown that discrete controllers achieve excellent performance in robotic navigation, often outperforming continuous controllers in a large variety of tasks [43].

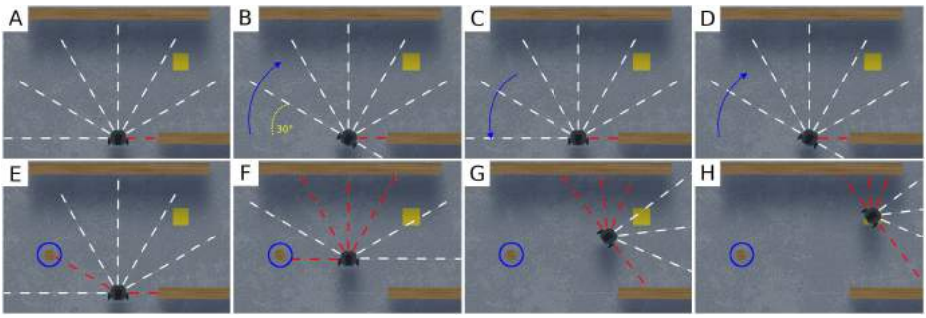


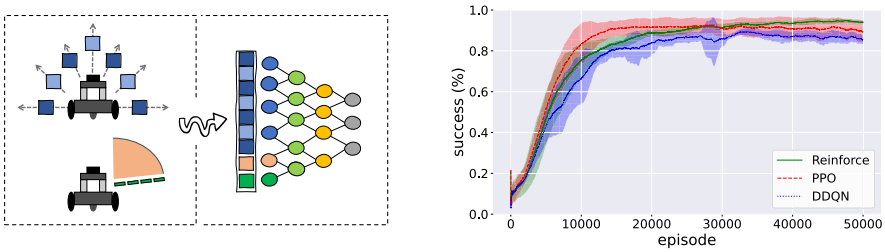
Fig. 3: An example of a simulated Turtlebot entering a 2-step loop. The white and red dashed lines represent the lidar beams (white indicates "clear", and red indicates that an obstacle is detected). The yellow square represents the target position; and the blue arrows indicate rotation. In the first row, from left to right, the Turtlebot is stuck in an infinite loop, alternating between right and left turns. Given the deterministic nature of the system, the agent will continue to select these same actions, ad infinitum. In the second row, from left to right, we present an almost identical configuration, but with an obstacle located 30° to the robot's left (circled in blue). The presence of the obstacle changes the input to the DNN, and allows the Turtlebot to avoid entering the infinite loop; instead, it successfully navigates to the target.

While the aforementioned DRL topology has been shown to be efficient for robotic navigation tasks, finding the optimal training algorithm and reward function is still an open problem. As part of our work, we trained multiple *deterministic* policies using the DRL algorithms presented in Section 2: DDQN [60], Reinforce [67], and PPO [50]. For the reward function, we used the following formulation:

$$\mathbb{R}_t = (d_{t-1} - d_t) \cdot \alpha - \beta,$$

where d_t is the distance from the target at time-step t; α is a normalization factor used to guarantee the stability of the gradient; and β is a fixed value, decreased at each time-step, and resulting in a total penalty proportional to the length of the path (by minimizing this penalty, the agent is encouraged to reach the target quickly). In our evaluation, we empirically selected $\alpha = 3$ and $\beta = 0.001$. Additionally, we added a final reward of $+1$ when the robot reached the target, or -1 in case it collided with an obstacle. For additional information regarding the training phase, see the full version of this paper [5].

DRL Training and Results. Using the training algorithms mentioned in Section 2, we trained a collection of DRL agents to solve the Turtlebot mapless navigation problem. We ran a stochastic training process, and thus obtained varied agents; of these, we only kept those that achieved a success rate of at least 96% during training. A total of 780 models were selected, consisting of 260 models per each of the three training algorithms. More specifically, for each



(a) The DRL controller (b) Average success rates

Fig. 4: (a) The DRL controller used for the robot in our case study. The DRL has nine input neurons: seven lidar sensor readings (blue), one input indicating the relative angle (orange) between the robot and the target, and one input indicating the distance (green) between the robot and the target. (b) The average success rates of models trained by each of the three DRL training algorithms, per training episode.

algorithm, all 260 models were generated from 52 random seeds. Each seed gave rise to a family of 5 models, where the individual family members differ in the number of training episodes used for training them. Fig. 4b shows the trained models' average success rate, for each algorithm used. We note that PPO was generally the fastest to achieve high accuracy. However, all three training algorithms successfully produced highly accurate agents.

4 Using Verification for Model Selection

All of our trained models achieved very high success rates, and so, at face value, there was no reason to favor one over the other. However, as we show next, a verification-based approach can expose multiple subtle differences between them. As our evaluation criteria, we define two properties of interest that are derived from the main goals of the robotic controller: (i) reaching the target; and (ii) avoiding collision with obstacles. Employing verification, we use these criteria to identify models that may fail to fulfill their goals, e.g., because they collide with various obstacles, are overly conservative, or may enter infinite loops without reaching the target. We now define the properties that we used, and the results of their verification are discussed in Section 5. Additional details regarding the precise encoding of our queries appear the full version of this paper [5].

Collision Avoidance. Collision avoidance is a fundamental and ubiquitous safety property [14] for navigation agents. In the context of Turtlebot, our goal is to check whether there exists a setting in which the robot is facing an obstacle, and chooses to move forward — even though it has at least one other viable option, in the form of a direction in which it is not blocked. In such situations, it is clearly preferable to choose to turn `LEFT` or `RIGHT` instead of choosing to move `FORWARD` and collide. See Fig. 5 for an illustration.

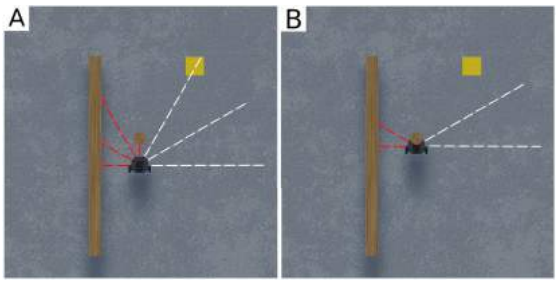


Fig. 5: Example of a single-step collision. The robot is not blocked on its right and can avoid the obstacle by turning (panel A), but it still chooses to move forward — and collides (panel B).

Given that turning `LEFT` or `RIGHT` produces an in-place rotation (i.e., the robot does not change its position), the only action that can cause a collision is `FORWARD`. In particular, a collision can happen when an obstacle is directly in front of the robot, or is slightly off to one side (just outside the front lidar's field of detection). More formally, we consider the safety property *"the robot does not collide at the next step"*, with three different types of collisions:

- `FORWARD COLLISION`: the robot detects an obstacle straight ahead, but nevertheless makes a step forward and collides with the obstacle.
- `LEFT COLLISION`: the robot detects an obstacle ahead and slightly shifted to the left (using the lidar beam that is 30° to the left of the one pointing straight ahead), but makes a single step forward and collides with the obstacle. The shape of the robot is such that in this setting, a collision is unavoidable.
- `RIGHT COLLISION`: the robot detects an obstacle ahead and slightly shifted to the right, but makes a single step forward and collides with the obstacle.

Recall that in mapless navigation, all observations are local — the robot has no sense of the global map, and can encounter any possible obstacle configuration (i.e., any possible sensor reading). Thus, in encoding these properties, we considered a single invocation of the DRL agent's DNN, with the following constraints:

1. All the sensors that are not in the direction of the obstacle receive a lidar input indicating that the robot can move either `LEFT` or `RIGHT` without risk of collision. This is encoded by lower-bounding these inputs.
2. The single input in the direction of the obstacle is upper-bounded by a value matching the representation of an obstacle, close enough to the robot so that it will collide if it makes a move `FORWARD`.
3. The input representing the distance to the target is lower-bounded, indicating that the target has not yet been reached (encouraging the agent to make a move).

The exact encoding of these properties is based on the physical characteristics of the robot and the lidar sensors, as explained in the full version of this paper [5].

Infinite Loops. Whereas collision avoidance is the natural safety property to verify in mapless navigation controllers, checking that progress is eventually made towards the target is the natural liveness property. Unfortunately, this property is difficult to formulate due to the absence of a complete map. Instead, we settle for a weaker property, and focus on verifying that the robot does not enter infinite loops (which would prevent it from ever reaching the target).

Unlike the case of collision avoidance, where a single step of the DRL agent could constitute a violation, here we need to reason about multiple consecutive invocations of the DRL controller, in order to identify infinite loops. This, again, is difficult to encode due to the absence of a global map, and so we focus on *in-place* loops: infinite sequences of steps in which the robot turns `LEFT` and `RIGHT`, but without ever moving `FORWARD`, thus maintaining its current location ad infinitum.

Our queries for identifying in-place loops encode that: (i) the robot does not reach the target in the first step; (ii) in the following k steps, the robot never moves `FORWARD`, i.e., it only performs turns; and (iii) the robot returns to an already-visited configuration, guaranteeing that the same behavior will be repeated by our deterministic agents. The various queries differ in the choice of k, as well as in the sequence of turns performed by the robot. Specifically, we encode queries for identifying the following kinds of loops:

- `ALTERNATING LOOP`: a loop where the robot performs an infinite sequence of $\langle$`LEFT`, `RIGHT`, `LEFT`, `RIGHT`, `LEFT`...$\rangle$ moves. A query for identifying this loop encodes $k = 2$ consecutive invocations of the DRL agent, after which the robot's sensors will again report the exact same reading, leading to an infinite loop. An example appears in Fig. 3. The encoding uses the "sliding window" principle, on which we elaborate later.
- `LEFT CYCLE`, `RIGHT CYCLE`: loops in which the robot performs an infinite sequence of $\langle$`LEFT`, `LEFT`, `LEFT`, ...$\rangle$ or $\langle$`RIGHT`, `RIGHT`, `RIGHT`, ...$\rangle$ operations accordingly. Because the Turtlebot turns at a 30° angle, this loop is encoded as a sequence of $k = 360°/30° = 12$ consecutive invocations of the DRL agent's DNN, all of which produce the same turning action (either `LEFT` or `RIGHT`). Using the sliding window principle guarantees that the robot returns to the same exact configuration after performing this loop, indicating that it will never perform any other action.

We also note that all the loop-identification queries include a condition for ensuring that the robot is not blocked from all directions. Consequently, any loops that are discovered demonstrate a clearly suboptimal behavior.

Specific Behavior Profiles. In our experiments, we noticed that the safe policies, i.e., the ones that do not cause the robot to collide, displayed a wide spectrum of different behaviors when navigating to the target. These differences occurred not only between policies that were trained by different algorithms, but also between policies trained by the same reward strategy — indicating that

these differences are, at least partially, due to the stochastic realization of the DRL training process.

Specifically, we noticed high variability in the length of the routes selected by the DRL policy in order to reach the given target: while some policies demonstrated short, efficient, paths that passed very close to obstacles, other policies demonstrated a much more conservative behavior, by selecting longer paths, and avoiding getting close to obstacles (an example appears in Fig. 6).

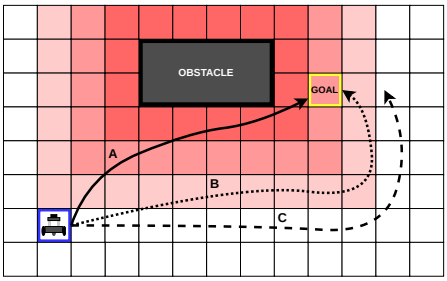


Fig. 6: Comparing paths selected by policies with different *bravery* levels. Path A takes the Turtlebot close to the obstacle (red area), and is the shortest. Path B maintains a greater distance from the obstacle (light red area), and is consequently longer. Finally, path C maintains such a significant distance from the obstacle (white area) that it is unable to reach the target.

Thus, we used our verification-driven approach to quantify how conservative the learned DRL agent is in the mapless navigation setting. Intuitively, a highly conservative policy will keep a significant safety margin from obstacles (possibly taking a longer route to reach its destination), whereas a "braver" and less conservative controller would risk venturing closer to obstacles. In the case of Turtlebot, the preferable DRL policies are the ones that guarantee the robot's safety (with respect to collision avoidance), and demonstrate a high level of bravery — as these policies tend to take shorter, optimized paths (see path A in Fig. 6), which lead to reduced energy consumption over the entire trail.

Bravery assessment is performed by encoding verification queries that identify situations in which the Turtlebot *can* move forward, but its control policy chooses not to. Specifically, we encode single invocations of the DRL model, in which we bound the lidar inputs to indicate that the Turtlebot is sufficiently distant from any obstacle and can safely move forward. We then use the verifier to determine whether, in this setting, a `FORWARD` output is possible. By altering and adjusting the bounds on the central lidar sensor, we can control how far away the robot perceives the obstacle to be. If we limit this distance to large values and the policy will still not move `FORWARD`, it is considered conservative; otherwise, it is considered brave. By conducting a binary search over these bounds [6], we can identify the shortest distance from an obstacle for which the policy *safely* orders the robot to move `FORWARD`. This value's inverse then serves as a bravery score for that policy.

Design-for-Verification: Sliding Windows. A significant challenge that we faced in encoding our verification properties, especially those that pertain to multiple consecutive invocations of the DRL policy, had to do with the local nature of the sensor readings that serve as input to the DNN. Specifically, if

the robot is in some initial configuration that leads to a sensor input x, and then chooses to move forward and reaches a successor configuration in which the sensor input is x', some connection between x and x' must be expressed as part of the verification query (i.e., nearby obstacles that exist in x cannot suddenly vanish in x'). In the absence of a global map, this is difficult to enforce.

In order to circumvent this difficulty, we used the *sliding window* principle, which has proven quite useful in similar settings [6, 21]. Intuitively, the idea is to focus on scenarios where the connections between x and x' are particularly straightforward to encode — in fact, most of the sensor information that appeared in x also appears in x'. This approach allows us to encode multistep queries, and is also beneficial in terms of performance: typically, adding sliding-window constraints reduces the search space explored by the verifier, and expedites solving the query.

In the Turtlebot setting, this is achieved by selecting a robot configuration in which the angle between two neighboring lidar sensors is identical to the turning angle of the robot (in our case, 30°). This guarantees, for example, that if the central lidar sensor observes an obstacle at distance d and the robot chooses to turn `RIGHT`, then at the next step, the lidar sensor just to the left of the central sensor must detect the same obstacle, at the same distance d. More generally, if at time-step t the 7 lidar readings (from left to right) are $\langle l_1, \ldots, l_7 \rangle$ and the robot turns `RIGHT`, then at time-step $t+1$ the 7 readings are $\langle l_2, l_3, \ldots, l_7, l_8 \rangle$, where only l_8 is a new reading. The case for a `LEFT` turn is symmetrical. By placing these constraints on consecutive states encountered by the robot, we were able to encode complex properties that involve multiple time-steps, e.g., as in the aforementioned infinite loops. An illustration appears in Fig. 3.

5 Experimental Evaluation

Next, we ran verification queries with the aforementioned properties, in order to assess the quality of our trained DRL policies. The results are reported below. In many cases, we discovered configurations in which the policies would cause the robot to collide or enter infinite loops; and we later validated the correctness of these results using a physical robot. We strongly encourage the reader to watch a short video clip that demonstrates some of these results [4]. Our code and benchmarks are also available online [3]. In our experiments, We used the *Marabou* verification engine [33] as our backend, although other engines could be used as well. For additional details regarding the experiments, we refer the reader to the full version of this paper [5].

Model Selection. In this set of experiments, we used verification to assess our trained models. Specifically, we used each of the three training algorithms (DDQN, Reinforce, PPO) to train 260 models, creating a total of 780 models. For each of these, we verified six properties of interest: three collision properties (`FORWARD COLLISION`, `LEFT COLLISION`, `RIGHT COLLISION`), and three loop properties (`ALTERNATING LOOP`, `LEFT CYCLE`, `RIGHT CYCLE`), as described in Section 4. This gives a total of 4680 verification queries. We ran all queries with a

	LEFT COLLISION		FORWARD COLLISION		RIGHT COLLISION	
Algorithm	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
DDQN	259	1	248	12	258	2
Reinforce	255	5	254	6	252	8
PPO	196	64	197	63	207	53

	ALTERNATING LOOP		LEFT CYCLE		RIGHT CYCLE		INSTABILITY
Algorithm	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT	# alternations
DDQN	260	0	56	77	56	61	21
Reinforce	145	115	5	185	120	97	10
PPO	214	45	26	198	30	198	1

Table 1: Results of the policy verification queries. We verified six properties over each of the 260 models trained per algorithm; SAT indicates that the property was violated, whereas UNSAT indicates that it held (to reduce clutter, we omit TIMEOUT and FAIL results). The rightmost column reports the stability values of the various training methods. For the full results see [3].

TIMEOUT value of 12 hours and a MEMOUT limit of $2G$; the results are summarized in Table 1. The single-step collision queries usually terminated within seconds, and the 2-step queries encoding an ALTERNATING LOOP usually terminated within minutes. The 12-step cycle queries, which are more complex, usually ran for a few hours. 9.6% of all queries hit the TIMEOUT limit (all from the 12-step cycle category), and none of the queries hit the MEMOUT limit.[2]

Our results exposed various differences between the trained models. Specifically, of the 780 models checked, 752 (over 96%) violated at least one of the single-step collision properties. These 752 collision-prone models include *all* 260 DDQN-trained models, 256 Reinforce models, and 236 PPO models. Furthermore, when we conducted a model filtering process based on all six properties (three collisions and three infinite loops), we discovered that 778 models out of the total of 780 (over 99.7%!) violated at least one property. The only two models that passed our filtering process were trained by the PPO algorithm.

Further analyzing the results, we observed that PPO models tended to be safer to use than those trained by other algorithms: they usually had the fewest violations per property. However, there are cases in which PPO proved less successful. For example, our results indicate that PPO-trained models are more prone to enter an ALTERNATING LOOP than those trained by Reinforce. Specifically, 214 (82.3%) of the PPO models have entered this undesired state, compared to 145 (55.8%) of the Reinforce models. We also point out that, similarly to the case with collision properties, *all* DDQN models violated this property.

Finally, when considering 12-step cycles (either LEFT CYCLE or RIGHT CYCLE), 44.8% of the DDQN models entered such cycles, compared to 30.7% of the Reinforce models, and just 12.4% of the PPO models. In computing these results, we

[2] We note that two queries failed due to internal errors in *Marabou*.

computed the fraction of violations (SAT queries) out of the number of queries that did not time out or fail, and aggregated SAT results for both cycle directions.

Interestingly, in some cases, we observed a bias toward violating a certain subcase of various properties. For example, in the case of entering full cycles — although 125 (out of 520) queries indicated that Reinforce-trained agents may enter a cycle in either direction, in 96% of these violations, the agent entered a RIGHT CYCLE. This bias is not present in models trained by the other algorithms, where the violations are roughly evenly divided between cycles in both directions.

We find that our results demonstrate that different "black-box" algorithms generalize very differently with respect to various properties. In our setting, PPO produces the safest models, while DDQN tends to produce models with a higher number of violations. We note that this does not necessarily indicate that PPO-trained models perform better, but rather that they are more robust to corner cases. Using our filtering mechanism, it is possible to select the safest models among the available, seemingly equivalent candidates.

Next, we used verification to compute the bravery score of the various models. Using a binary search, we computed for each model the minimal distance a dead-ahead obstacle needs to have for the robot to *safely* move forward. The search range was $[0.18, 1]$ meters, and the optimal values were computed up to a 0.01 precision (see the full version of this paper [5] for additional details). Almost all binary searches terminated within minutes, and none hit the TIMEOUT threshold.

By first filtering the models based on their safe behavior, and then by their bravery scores, we are able to find the few models that are both safe (do not collide), and not overly conservative. These models tend to take efficient paths, and may come close to an obstacle, but without colliding with it. We also point out that over-conservativeness may significantly reduce the success rate in specific scenarios, such as cases in which the obstacle is close to the target. Specifically, of the only two models that survived the first filtering stage, one is considerably more conservative than the other — requiring the obstacle to be twice as distant as the other, braver, model requires it to be, before moving forward.

Algorithm Stability Analysis. As part of our experiments, we used our method to assess the three training algorithms — DDQN, PPO, and Reinforce. Recall that we used each algorithm to train 52 families of 5 models each, in which the models from the same family are generated from the same random seed, but with a different number of training iterations. While all models obtained a high success rate, we wanted to check how often it occurred that a model successfully learned to satisfy a desirable property after some training iterations, only to forget it after additional iterations. Specifically, we focused on the 12-step full-cycle properties (LEFT CYCLE and RIGHT CYCLE), and for each family of 5 models checked whether some models satisfied the property while others did not.

We define a family of models to be *unstable* in the case where a property holds in the family, but ceases to hold for another model from the same family with a higher number of training iterations. Intuitively, this means that the model "forgot" a desirable property as training progressed. The *instability value* of each algorithm type is defined to be the number of unstable 5-member families.

Although all three algorithms produced highly accurate models, they displayed significant differences in the stability of their produced policies, as can be seen in the rightmost column of Table 1. Recall that we trained 52 families of models using each algorithm, and then tested their stability with respect to two properties (corresponding to the two full cycle types). Of these, the DDQN models display 21 *unstable* alternations — more than twice the number of alterations demonstrated by Reinforce models (10), and significantly higher than the number of alternations observed among the PPO models (1).

These results shed light on the nature of these training algorithms — indicating that DDQN is a significantly less stable training algorithm, compared to PPO and Reinforce. This is in line with previous observations in non-verification-related research [50], and is not surprising, as the primary objective of PPO is to limit the changes the optimizer performs between consecutive training iterations.

Gradient-Based Methods. We also conducted a thorough comparison between our verification-based approach and competing gradient-based methods. Although gradient-based attacks are extremely scalable, our results (summarized in [5]) show that they may miss many of the violations found by our complete, verification-based procedure. For example, when searching for collisions, our approach discovered a total of 2126 `SAT` results, while the gradient-based method discovered only 1421 `SAT` results — a 33% decrease (!). In addition, given that gradient-based methods are unable to return `UNSAT`, they are also incapable of proving that a property always holds, and hence cannot formally guarantee the safety of a policy in question. Thus, performing model selection based on gradient-based methods could lead to skewed results. We refer the reader to the full version of this paper [5], in which we elaborate on gradient attacks and the experiments we ran, demonstrating the advantages of our approach for model selection, when compared to gradient-based methods.

6 Related Work

Due to the increasing popularity of DNNs, the formal methods community has put forward a plethora of tools and approaches for verifying DNN correctness [20,24,26,28,31–33,36,39,52,59]. Recently, the verification of systems involving multiple DNN invocations, as well as hybrid systems with DNN components, has been receiving significant attention [6,9,17,18,22,34,54,61]. Our work here is another step toward applying DNN verification techniques to additional, real-world systems and properties of interest.

In the robotics domain, multiple approaches exist for increasing the reliability of learning-based systems [48,62,69]; however, these methods are mostly heuristic in nature [1,23,42]. To date, existing techniques rely mostly on Lagrangian multipliers [38,49,53], and do not provide formal safety guarantees; rather, they optimize the training in an attempt to learn the required policies [12]. Other, more formal approaches focus solely on the systems' input-output relations [15,41], without considering multiple invocations of the agent and its interactions with

the environment. Thus, existing methods are not able to provide rigorous guarantees regarding the correctness of multistep robotic systems, and do not take into account sequential decision making — which renders them insufficient for detecting various safety and liveness violations.

Our approach is orthogonal and complementary to many existing safe DRL techniques. Reward reshaping and shielding techniques (e.g., [2]) improve safety by altering the training loop, but typically afford no formal guarantees. Our approach can be used to complement them, by selecting the most suitable policy from a pool of candidates, post-training. Guard rules and runtime shields are beneficial for preventing undesirable behavior of a DNN agent, but are sometimes less suited for specifying the *desired* actions it should take instead. In contrast, our approach allows selecting the optimal policy from a pool of candidates, without altering its decision-making.

7 Conclusion

Through the case study described in this paper, we demonstrate that current verification technology is applicable to real-world systems. We show this by applying verification techniques for improving the navigation of DRL-based robotic systems. We demonstrate how off-the-shelf verification engines can be used to conduct effective model selection, as well as gain insights into the stability of state-of-the-art training algorithms. As far as we are aware, ours is the first work to demonstrate the use of formal verification techniques on multistep properties of actual, real-world robotic navigation platforms. We also believe the techniques developed here will allow the use of verification to improve additional multistep systems (autonomous vehicles, surgery-aiding robots, etc.), in which we can impose a transition function between subsequent steps. However, our approach is limited by DNN-verification technology, which we use as a black-box backend. As that technology becomes more scalable, so will our approach. Moving forward, we plan to generalize our work to richer environments — such as cases where a memory-enhanced agent interacts with moving objects, or even with multiple agents in the same arena, as well as running additional experiments with deeper networks, and more complex DRL systems. In addition, we see probabilistic verification of stochastic policies as interesting future work.

Acknowledgements. The work of Amir, Yerushalmi and Katz was partially supported by the Israel Science Foundation (grant number 683/18). The work of Amir was supported by a scholarship from the Clore Israel Foundation. The work of Corsi, Marzari, and Farinelli was partially supported by the "Dipartimenti di Eccellenza 2018-2022" project, funded by the Italian Ministry of Education, Universities, and Research (MIUR). The work of Yerushalmi and Harel was partially supported by a research grant from the Estate of Harry Levine, the Estate of Avraham Rothstein, Brenda Gruss and Daniel Hirsch, the One8 Foundation, Rina Mayer, Maurice Levy, and the Estate of Bernice Bernath, grant 3698/21 from the ISF-NSFC (joint to the Israel Science Foundation and the National

Science Foundation of China), and a grant from the Minerva foundation. We thank Idan Refaeli for his contribution to this project.

References

1. J. Achiam, D. Held, A. Tamar, and P. Abbeel. Constrained Policy Optimization. In *Proc. 34th Int. Conf. on Machine Learning (ICML)*, pages 22–31, 2017.
2. M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu. Safe Reinforcement Learning via Shielding. In *Proc. 32th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 2669–2678, 2018.
3. G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Supplementary Artifact, 2022. https://doi.org/10.5281/zenodo.7496352.
4. G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Supplementary Video, 2022. https://youtu.be/QIZqOgxLkAE.
5. G. Amir, D. Corsi, R. Yerushalmi, L. Marzari, D. Harel, A. Farinelli, and G. Katz. Verifying Learning-Based Robotic Navigation Systems, 2023. Technical Report. https://arxiv.org/abs/2205.13536.
6. G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.
7. R. Amsters and P. Slaets. Turtlebot 3 as a Robotics Education Platform. In *Proc. 10th Int. Conf. on Robotics in Education (RiE)*, pages 170–181, 2019.
8. G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.
9. E. Bacci, M. Giacobbe, and D. Parker. Verifying Reinforcement Learning Up to Infinity. In *Proc. 30th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2021.
10. T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks and its Security Applications. In *Proc. ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, pages 1249–1264, 2019.
11. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. http://arxiv.org/abs/1604.07316.
12. L. Brunke, M. Greeff, A. Hall, Z. Yuan, S. Zhou, J. Panerati, and A. Schoellig. Safe Learning in Robotics: From Learning-Based Control to Safe Reinforcement Learning. *Annual Review of Control, Robotics, and Autonomous Systems*, 5, 2021.
13. H. Chiang, A. Faust, M. Fiser, and A. Francis. Learning Navigation Behaviors End-to-End with AutoRL. *IEEE Robotics and Automation Letters (RA-L/ICRA)*, 4(2):2007–2014, 2019.
14. E. Clarke, T. Henzinger, H. Veith, and R. Bloem. *Handbook of Model Checking*, volume 10. Springer, 2018.
15. D. Corsi, E. Marchesini, and A. Farinelli. Formal Verification of Neural Networks for Safety-Critical Tasks in Deep Reinforcement Learning. In *Proc. 37th Conf. on Uncertainty in Artificial Intelligence (UAI)*, pages 333–343, 2021.
16. L. Deng and Y. Liu. *Deep Learning in Natural Language Processing*. Springer, 2018.

17. S. Dutta, X. Chen, and S. Sankaranarayanan. Reachability Analysis for Neural Feedback Systems using Regressive Polynomial Rule Inference. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, pages 157–168, 2019.
18. S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Learning and Verification of Feedback Control Systems using Feedforward Neural Networks. *IFAC-PapersOnLine*, 51(16):151–156, 2018.
19. S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output Range Analysis for Deep Feedforward Neural Networks. In *Proc. 10th NASA Formal Methods Symposium (NFM)*, pages 121–138, 2018.
20. R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.
21. T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 305–318, 2021.
22. N. Fulton and A. Platzer. Safe Reinforcement Learning via Formal Methods: Toward Safe Control through Proof and Learning. In *Proc. 32nd AAAI Conf. on Artificial Intelligence (AAAI)*, 2018.
23. J. Garcıa and F. Fernández. A Comprehensive Survey on Safe Reinforcement Learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.
24. T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.
25. I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
26. D. Gopinath, G. Katz, C. Păsăreanu, and C. Barrett. DeepSafe: A Data-driven Approach for Assessing Robustness of Neural Networks. In *Proc. 16th. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 3–19, 2018.
27. D. Gunning. Explainable Artificial Intelligence (XAI), 2017. Defense Advanced Research Projects Agency (DARPA) Project.
28. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.
29. R. Ivanov, T. Carpenter, J. Weimer, R. Alur, G. Pappas, and I. Lee. Verifying the Safety of Autonomous Systems with Neural Network Controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(1):1–26, 2020.
30. P. Jin, J. Tian, D. Zhi, X. Wen, and M. Zhang. Trainify: A CEGAR-Driven Training and Verification Framework for Safe Deep Reinforcement Learning. In *Proc. 34th Int. Conf. on Computer Aided Verification (CAV)*, pages 193–218, 2022.
31. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 97–117, 2017.
32. G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021.
33. G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Frame-

work for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.
34. Y. Kazak, C. Barrett, G. Katz, and M. Schapira. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 83–89, 2019.
35. B. Könighofer, F. Lorber, N. Jansen, and R. Bloem. Shield Synthesis for Reinforcement Learning. In *Proc. Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 290–306, 2020.
36. L. Kuper, G. Katz, J. Gottschlich, K. Julian, C. Barrett, and M. Kochenderfer. Toward Scalable Verification for Safety-Critical Deep Networks, 2018. Technical Report. https://arxiv.org/abs/1801.05950.
37. Y. Li. Deep Reinforcement Learning: An Overview, 2017. Technical Report. http://arxiv.org/abs/1701.07274.
38. Y. Liu, J. Ding, and X. Liu. Ipo: Interior-Point Policy Optimization under Constraints. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 4940–4947, 2020.
39. A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1706.07351.
40. Z. Lyu, C. Y. Ko, Z. Kong, N. Wong, D. Lin, and L. Daniel. Fastened Crown: Tightened Neural Network Robustness Certificates. In *Proc. 34th AAAI Conf. on Artificial Intelligence (AAAI)*, pages 5037–5044, 2020.
41. E. Marchesini, D. Corsi, and A. Farinelli. Benchmarking Safe Deep Reinforcement Learning in Aquatic Navigation. In *Proc. IEEE/RSJ Int. Conf on Intelligent Robots and Systems (IROS)*, 2021.
42. E. Marchesini, D. Corsi, and A. Farinelli. Exploring Safer Behaviors for Deep Reinforcement Learning. In *Proc. 35th AAAI Conf. on Artificial Intelligence (AAAI)*, 2021.
43. E. Marchesini and A. Farinelli. Discrete Deep Reinforcement Learning for Mapless Navigation. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 10688–10694, 2020.
44. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning, 2013. Technical Report. https://arxiv.org/abs/1312.5602.
45. S. M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard. Universal Adversarial Perturbations. In *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 1765–1773, 2017.
46. C. Nandkumar, P. Shukla, and V. Varma. Simulation of Indoor Localization and Navigation of Turtlebot 3 using Real Time Object Detection. In *Proc. Int. Conf. on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, 2021.
47. M. Pfeiffer, S. Shukla, M. Turchetta, C. Cadena, A. Krause, R. Siegwart, and J. Nieto. Reinforced Imitation: Sample Efficient Deep Reinforcement Learning for Mapless Navigation by Leveraging Prior Demonstrations. *IEEE Robotics and Automation Letters*, 3(4):4423–4430, 2018.
48. A. Ray, J. Achiam, and D. Amodei. Benchmarking Safe Exploration in Deep Reinforcement Learning, 2019. Technical Report. https://cdn.openai.com/safexp-short.pdf.
49. J. Roy, R. Girgis, J. Romoff, P. Bacon, and C. Pal. Direct Behavior Specification via Constrained Reinforcement Learning, 2021. Technical Report. https://arxiv.org/abs/2112.12228.

50. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms, 2017. Technical Report. http://arxiv.org/abs/1707.06347.
51. K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014. Technical Report. http://arxiv.org/abs/1409.1556.
52. G. Singh, T. Gehr, M. Puschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.
53. A. Stooke, J. Achiam, and P. Abbeel. Responsive Safety in Reinforcement Learning by Pid Lagrangian Methods. In *Proc. 37th Int. Conf. on Machine Learning (ICML)*, pages 9133–9143, 2020.
54. X. Sun, H. Khedr, and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.
55. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.
56. R. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 1999.
57. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. http://arxiv.org/abs/1312.6199.
58. L. Tai, G. Paolo, and M. Liu. Virtual-to-Real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 31–36, 2017.
59. V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, 2017. Technical Report. http://arxiv.org/abs/1711.07356.
60. H. Van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-Learning. In *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI)*, 2016.
61. M. Vasić, A. Petrović, K. Wang, M. Nikolić, R. Singh, and S. Khurshid. MoËT: Mixture of Expert Trees and its Application to Verifiable Reinforcement Learning. *Neural Networks*, 151:34–47, 2022.
62. A. Wachi and Y. Sui. Safe Reinforcement Learning in Constrained Markov Decision Processes. In *Proc. 37th Int. Conf. on Machine Learning (ICML)*, pages 9797–9806, 2020.
63. A. Wahid, A. Toshev, M. Fiser, and T. Lee. Long Range Neural Navigation Policies for the Real World. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 82–89, 2019.
64. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, pages 1599–1614, 2018.
65. K. Yoneda, H. Tehrani, T. Ogawa, N. Hukuyama, and S. Mita. Lidar Scan Feature for Localization with Highly Precise 3-D Map. In *Proc. IEEE Intelligent Vehicles Symposium (IV)*, pages 1345–1350, 2014.
66. H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.

67. J. Zhang, J. Kim, B. O'Donoghue, and S. Boyd. Sample Efficient Reinforcement Learning with REINFORCE, 2020. Technical Report. https://arxiv.org/abs/2010.11364.
68. J. Zhang, J. Springenberg, J. Boedecker, and W. Burgard. Deep Reinforcement Learning with Successor Features for Navigation across Similar Environments. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2017.
69. L. Zhang, R. Zhang, T. Wu, R. Weng, M. Han, and Y. Zhao. Safe Reinforcement Learning with Stability Guarantee for Motion Planning of Autonomous Vehicles. *IEEE Transactions on Neural Networks and Learning Systems*, 32(12):5435–5444, 2021.
70. O. Zhelo, J. Zhang, L. Tai, M. Liu, and W. Burgard. Curiosity-Driven Exploration for Mapless Navigation with Deep Reinforcement Learning, 2018. Technical Report. https://arxiv.org/abs/1804.00456.

Make Flows Small Again: Revisiting the Flow Framework

Roland Meyer[1], Thomas Wies[2], and Sebastian Wolff[2(✉)]

[1] TU Braunschweig, Braunschweig, Germany, roland.meyer@tu-bs.de
[2] New York University, New York, USA, {wies,sebastian.wolff}@cs.nyu.edu

Abstract We present a new flow framework for separation logic reasoning about programs that manipulate general graphs. The framework overcomes problems in earlier developments: it is based on standard fixed point theory, guarantees least flows, rules out vanishing flows, and has an easy to understand notion of footprint as needed for soundness of the frame rule. In addition, we present algorithms for automating the frame rule, which we evaluate on graph updates extracted from linearizability proofs for concurrent data structures. The evaluation demonstrates that our algorithms help to automate key aspects of these proofs that have previously relied on user guidance or heuristics.

Keywords: Separation Logic · Graph Algorithms · Frame Inference.

1 Introduction

The flow framework [23,24] is an abstraction mechanism based on separation logic [5, 32, 40] that enables reasoning about global inductive invariants of general graphs in a local manner. The framework has proved useful to verify intricate algorithms that are difficult to handle by other techniques, such as the Priority Inheritance Protocol, object-oriented design patterns, and complex concurrent data structures [22,24,27,34]. However, these efforts have also exposed some rough corners in the underlying meta theory that either limit expressivity or automation. In this paper, we propose a new meta theory for the flow framework that aims to strike a balance between these conflicting requirements. In addition, we present algorithms that aid proof automation.

Background. The central notion of the flow framework is that of a *flow*. Given a commutative monoid $(\mathbb{M}, +, 0)$ (e.g. natural numbers with addition), and a graph with nodes X and an *edge function* $E\colon X \times X \to \mathbb{M} \to \mathbb{M}$, a flow is a function $\mathit{fl}\colon X \to \mathbb{M}$ that satisfies the *flow equation*:

$$\forall x \in X.\quad \mathit{fl}(x) = \mathit{in}_x + \textstyle\sum_{y \in X} E_{(y,x)}(\mathit{fl}(y)) \ .$$

That is, fl is a fixed point of the function that assigns every node x an initial value $\mathit{in}_x \in \mathbb{M}$, its *inflow*, and then propagates these values through the graph according to the edge function. This is akin to a forward data flow analysis where the monoid operation $+$ is used as the join. By choosing an appropriate flow monoid, inflow, and edge function, one can express inductive properties of graphs (reachability, sortedness, etc.) in terms of conditions that refer only to each node's flow value $\mathit{fl}(x)$.

A graph endowed with an inflow and associated flow is a *flow graph*. An example flow graph h is shown on the right-hand side of Fig. 1a. Here, the flow value $\mathit{fl}(w)$ for

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 628–646, 2023.
https://doi.org/10.1007/978-3-031-30823-9_32

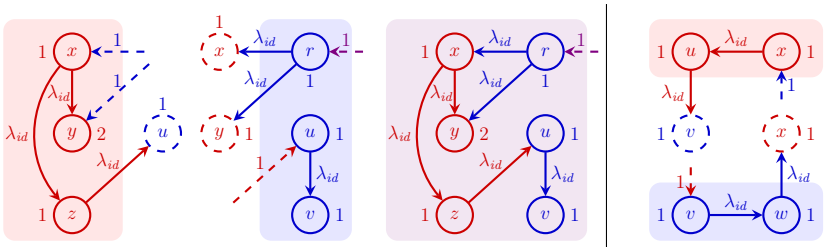


Figure 1. (a) Two flow graphs h_1 with nodes $h_1.X = \{ x, y, z \}$ (left) and h_2 with nodes $h_2.X = \{ r, u, v \}$ (center) for the flow monoid of natural numbers with addition. The edge label λ_{id} stands for the identity function. Omitted edges are labeled by the constant 0 function. Dashed edges represent the inflows. Nodes are labeled by their flow, respectively, outflow. The right side shows the composition $h = h_1 * h_2$. **(b)** Two flow graphs h_1 with $h_1.X = \{ u, x \}$ (top) and h_2 with $h_2.X = \{ v, w \}$ (bottom) whose composition is undefined due to vanishing flows.

a node w counts the number of paths from r to w. A flow graph can be partial and have edges to nodes outside of X like the node u for h_1 in Fig. 1a. If we include these nodes in the computation of the flow, then their flow values constitute the *outflow* of the flow graph. For instance, the outflow of h_1 for u is 1.

Flow graphs are equipped with a notion of disjoint composition, $h = h_1 * h_2$. An example is given in Fig. 1a. The composition is only defined if the union of the flows of h_1 and h_2 is again a flow of h. This may not always be the case. For instance, the inflows and outflows of h_1 and h_2 may be mutually incompatible such as h_1 sending outflow 2 to u whereas the inflow to u in h_2 is only 1.

Flow graph composition yields a *separation algebra*. That is, if we use flow graphs as an abstraction of program states (e.g., the heap), then we can use separation logic to reason locally about properties of programs that are expressed in terms of the induced flow graphs. For example, suppose the program updates the flow graph h in Fig. 1a to a new flow graph h' by inserting a new edge labeled λ_{id} between the nodes r and u. This increases the flow of u and v from 1 to 2. We can break this update down as follows. First, we decompose h into h_1 and h_2. Next, we obtain h_2' from h_2 by inserting the edge and updating the flow of u and v to 2. Finally, we compose h_2' again with h_1 to obtain h'. Note that the composition $h_1 * h_2'$ is still defined. This means that any property expressed over the flow in the h_1-portion of h still holds in h'. This is the well-known *frame rule* of separation logic, instantiated for flow graphs.

The crux in applying the frame rule is to show that the composition $h_1 * h_2'$ is indeed defined. One can do this locally by showing that the update $h_2 \rightsquigarrow h_2'$ is *frame-preserving*, i.e., for *any* h_1 such that $h_1 * h_2$ is defined, $h_1 * h_2'$ is also defined.

Typically, the flow subgraphs involved in a frame-preserving update $h_2 \rightsquigarrow h_2'$ include more nodes than those immediately affected by the update. For instance, consider the subgraphs of h and h' in our example that consist only of the nodes $\{r, u\}$ directly affected by inserting the edge. These subgraphs do not constitute a frame-preserving update because inserting the edge between r and u also changes the outflow to v from

1 to 2. Hence, the updated subgraph for $\{r, u\}$ would no longer compose with the rest of h where v's flow is still 1 instead of 2. We refer to a set of nodes such as $\{r, u, v\}$ that identifies a frame-preserving update as the update's *footprint*.

Meta theories of flow graphs. In addition to ensuring that flow graph composition yields a separation algebra, there are two desiderata that one has to take into consideration when designing a meta theory of flow graphs:

- *Obtaining unique flows.* When encoding inductive properties using flows, one is often interested in a particular flow, most commonly the least fixed point of the flow equation for a given inflow. One therefore needs a way to focus the reasoning on the particular flow of interest.
- *Identifying frame-preserving updates.* In order to enable the application of the frame rule, one needs a way to effectively compute candidate footprints and check whether they identify frame-preserving updates.

The first subgoal is crucial for expressivity and the second one for proof automation. Achieving one subgoals makes it more difficult to achieve the other. Specifically, consider the meta theory proposed in [24]. It requires that the flow monoid $(\mathbb{M}, +, 0)$ is also cancellative ($m + n_1 = o$ and $m + n_2 = o$ implies $n_1 = n_2$). Requiring cancellativity has the advantage that it is easy to check if an update $h \rightsquigarrow h'$ is frame-preserving: it suffices to show that h and h' have the same inflow and outflow. Cancellativity also ensures that for each flow fl, there exists a unique inflow that produces fl. Hence, it is sufficient to track only fl since the inflow is a derived quantity. However, the converse does not hold.

In fact, obtaining unique flows for cancellative $\mathbb{M}$ becomes more difficult. A natural requirement that one would like to impose on $\mathbb{M}$ is that the pre-order induced by $+$ forms a complete partial order (cpo) or even a complete lattice. This way, one can focus on the least flow, which is guaranteed to exist if one applies standard fixed point theorems, imposing only mild assumptions on the edge functions. However, cancellativity is inherently incompatible with standard domain-theoretic prerequisites. For instance, the only ordered cancellative commutative monoid that is a directed cpo is the trivial one: $\mathbb{M}_0 = \{0\}$. Similarly, $\mathbb{M}_0$ is the only such monoid that has a greatest element.

For cases where unique flows are desired, [24] imposes additional requirements on the edge functions (nil-potent) or the graph structure (effectively acyclic). The former is quite restrictive in terms of expressivity. The latter again complicates the computation of frame-preserving updates: one now has to ensure that no cycles are introduced when the updated graph h_2' is composed with its frame h_1. In fact, for the effectively acyclic case, [24] only provides a sufficient condition that a given footprint yields a frame-preserving update but it gives no algorithm for computing such a footprint.

Contributions. In this paper, we propose a new meta theory of flows based on flow monoids that form ω-cpos (but need not be cancellative). The cpo requirement yields the desired least fixed point semantics. The differences in the requirements on the flow monoid necessitate a new notion of flow graph composition. In particular, for a least fixed point semantics of flows, $h = h_1 * h_2$ is only defined if the flows of h_1 and h_2 do not vanish. An example of such a situation is shown in Fig. 1b, where the flows in h_1 and h_2 would vanish to 0 in $h_1 * h_2$ because the created cycle has no external inflow. Moreover, an update $h \rightsquigarrow h'$ is frame-preserving if h and h' route inflows to outflows in the same way. We formalize this condition using a notion of contextual equivalence

of the graphs' *transfer functions*, which are the least fixed points of the flow equation, parameterized by the inflows and restricted to the nodes outside the graphs. We then identify conditions on the edge functions that are commonly satisfied in practice and that allow us to effectively check contextual equivalence of transfer functions. This result is remarkable because the flow monoid can have infinite ascending chains and the flow graphs can be cyclic. Building on this equivalence check, we propose an iterative algorithm for computing footprints of updates. This algorithm enables the automation of the frame rule for reasoning about programs manipulating flow graphs. We evaluate the presented algorithms on a benchmark suite of flow graph updates that are extracted from linearizability proofs for concurrent search structures constructed by the tool `plankton` [26,27]. The evaluation demonstrates that our algorithms help to automate key aspects of these proofs that have previously relied on user guidance or heuristics.

2 Flow Graph Separation Algebra

We start with the presentation of our new separation algebra of flow graphs.

Given a commutative monoid $(\mathbb{M}, +, 0)$, we define the binary relation $\leq$ on $\mathbb{M}$ by $n \leq m$ if there is $o \in \mathbb{M}$ with $m = n + o$. Flow values are drawn from a *flow monoid*, a commutative monoid for which the relation $\leq$ is an ω-cpo. That is, $\leq$ is a partial order and every ascending chain $K = m_0 \leq m_1 \leq \ldots$ in $\mathbb{M}$ has a least upper bound, denoted $\bigsqcup K$. We expect $n + \bigsqcup K = \bigsqcup(n + K)$. In the following, we fix a flow monoid $(\mathbb{M}, +, 0)$.

Let $\mathit{ContFun}(\mathbb{M} \to \mathbb{M})$ be the continuous functions in $\mathbb{M} \to \mathbb{M}$. Recall that a function $f : \mathbb{M} \to \mathbb{M}$ is *continuous* [43] if it commutes with limits of ascending chains, $f(\bigsqcup K) = \bigsqcup f(K)$ for every chain K in $\mathbb{M}$. We lift $+$ and $\leq$ to functions $\mathbb{M} \to \mathbb{M}$ in the expected way. An empty iterated sum $\sum_{i \in \varnothing} m_i$ is defined to be 0.

Lemma 1. *$(\mathit{ContFun}(\mathbb{M} \to \mathbb{M}), \circ, \mathit{id})$ is a monoid. Moreover, if $(\mathbb{M}, \leq)$ is an ω-cpo, so is $(\mathit{ContFun}(\mathbb{M} \to \mathbb{M}), \leq)$.*

A *flow graph* is a tuple $h = (X, E, \mathit{in})$ consisting of a finite set of nodes $X \subseteq \mathbb{N}$, a set of edges $E : X \times \mathbb{N} \to \mathit{ContFun}(\mathbb{M} \to \mathbb{M})$ labeled by continuous functions, and an *inflow* $\mathit{in} : (\mathbb{N} \setminus X) \times X \to \mathbb{M}$. We use FG for the set of all flow graphs and denote the empty flow graph by $h_\varnothing \triangleq (\varnothing, \varnothing, \varnothing)$.

We define two derived functions for flow graphs. First, the *flow* is the least function $\mathit{flow} : X \to \mathbb{M}$ satisfying the flow equation: $\mathit{flow}(x) = \mathit{in}_x + \mathit{rhs}_x(\mathit{flow})$, for all $x \in X$. Here, $\mathit{in}_x \triangleq \sum_{y \in (\mathbb{N} \setminus X)} \mathit{in}(y, x)$ is a monoid value and $\mathit{rhs}_x \triangleq \sum_{y \in X} E_{(y,x)}$ is a function of type $\mathit{ContFun}((X \to \mathbb{M}) \to \mathbb{M})$. Finally, we also define the *outflow* $\mathit{out} : X \times (\mathbb{N} \setminus X) \to \mathbb{M}$ by $\mathit{out}(x, y) \triangleq E_{(x,y)}(\mathit{flow}(x))$.

Example 1. For linearizability proofs of concurrent search structures one can use a flow that labels every data structure node x with its *inset*, the set of keys k' such that a thread searching for k' may traverse the node x [22,23]. Translated to our setting, the relevant flow monoid is the powerset of keys, $\mathbb{P}(\mathbb{Z} \cup \{-\infty, \infty\})$, with set union as addition. Figure 2 shows two keyset flow graphs that abstract potential states of a concurrent set implementation based on sorted linked lists. When a key k is removed from the set, the node x that stores k is first marked to indicate that x has been logically deleted. In

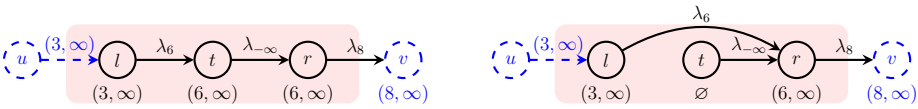


Figure 2. Two flow graphs h_1 (left) and h_2 (right) with $h_1.X = h_2.X = \{ l, t, r \}$ for the keyset flow monoid $\mathbb{P}(\mathbb{Z} \cup \{ -\infty, \infty \})$. The edge label λ_k for a key k denotes the function $\lambda m.\, (m \setminus [-\infty, k])$.

a second step, x is then physically unlinked from the list. The idea of the abstraction is that an edge leaving a node x that stores a key k is labeled by the function λ_k if x is unmarked and otherwise by $\lambda_{-\infty}$. This is because a search for $k' \in \mathbb{Z}$ will traverse the edge leaving x iff $k < k'$ or x is marked. In the figure, l and r are assumed to be unmarked, storing keys 6 and 8, respectively. Node t is assumed to be marked. Flow graph h_2 is obtained from h_1 by physically unlinking the marked node t. Using the keyset flow one can then express the crucial data structure invariants that are needed for a linearizability proof based on local reasoning (e.g., the invariant that the logical contents of a node is always a subset of its inset).

We note that the inflow of the global flow graph that abstracts the program state can be used in the specification. In the example, one lets $in_r = \mathbb{Z}$ for the root r of the data structure and $in_x = \varnothing$ for all other nodes to indicate that all searches start at r. □

Composition without vanishing flows. To define the composition of flow graphs, $h_1 * h_2$, we proceed in two steps. We first define an auxiliary composition that may suffer from *vanishing* flows, local flows that disappear in the composition. That is, this composition is defined for the flow graphs shown in Fig. 1b. In the composed graph the flow of each node is 0 where it was 1 before the composition—the flow vanishes. This means that the auxiliary composition does not allow to lift lower bounds on the flow values from the individual components to the composed graph. Hence, the actual composition restricts the auxiliary composition to rule out such vanishing flows. Definedness of the auxiliary composition requires disjointness of the nodes in h_1 and h_2. Moreover, the outflow of one flow graph has to match the inflow expectations of the other:

$$h_1 \#\# h_2 \quad \text{if} \quad X_1 \cap X_2 = \varnothing \;\wedge\; \forall x \in X_1,\, y \in X_2.\; out_1(x,y) = in_2(x,y) \wedge out_2(y,x) = in_1(y,x)\,.$$

The auxiliary composition $h_1 \uplus h_2$ removes the inflow provided by the other component:

$$h_1 \uplus h_2 \quad \triangleq \quad (X_1 \uplus X_2, E_1 \uplus E_2, (in_1 \uplus in_2)|_{(\mathbb{N} \setminus (X_1 \uplus X_2)) \times (X_1 \uplus X_2)})\,.$$

To rule out vanishing flows, we incorporate a suitable equality on the flows:

$$h_1 \# h_2 \quad \text{if} \quad h_1 \#\# h_2 \;\wedge\; h_1.\mathit{flow} \uplus h_2.\mathit{flow} = (h_1 \uplus h_2).\mathit{flow}\,.$$

Only if the latter equality holds, do we have the composition $h_1 * h_2 \triangleq h_1 \uplus h_2$. It is worth noting that $h_1.\mathit{flow} \uplus h_2.\mathit{flow} \geq (h_1 \uplus h_2).\mathit{flow}$ always holds. What definedness really asks for is the reverse inequality.

Recall from [5] that a *separation algebra* is a partial commutative monoid $(\Sigma, *, \mathsf{emp})$ with a set of units $\mathsf{emp} \subseteq \Sigma$.

Lemma 2. *$(FG, *, \{ h_\varnothing \})$ is a separation algebra.*

3 Frame-Preserving Updates

Since flow graphs form a separation algebra, we can use separation logic assertions to describe sets of flow graphs as in [24] and then use them to prove separation logic Hoare triples. A key proof rule used in such proofs is the frame rule. Given separation logic assertions P_1 and P_2, and a command c, the frame rule states: if the Hoare triple $\{P_1\}\, c\, \{P_2\}$ is valid, then so is $\{P_1 * F\}\, c\, \{P_2 * F\}$ for any *frame* F. The remainder of the paper focuses on developing algorithms for automating this proof rule.

The flow graphs described by an assertion may have unbounded size (e.g., due to the use of *iterated separating conjunctions*). We only consider bounded flow graphs in the following; the unbounded case is known to be a challenge for which orthogonal techniques are being developed (cf. Sect. 6). However, even if the flow graphs have bounded size, there may still be infinitely many of them because the inflows and edge functions are encoded symbolically in a logical theory of the flow monoid. For pedagogy, we present our algorithms in terms of concrete flow graphs rather than symbolic ones. However, our development readily extends to symbolic representations assuming the underlying flow monoid theory is decidable. In fact, our implementation discussed in Sect. 5 works with symbolic flow graphs.

The soundness of the frame rule relies on the assumption that the state update induced by the command c satisfies a certain locality condition. In our setting, this condition amounts to checking that the update of P_1 under c is *frame-preserving* with respect to flow graph composition. For the flow graphs h_1 described by P_1 and all flow graphs h_2 in the post image of h_1 under c, this means that $h_1 \# h$ implies $h_2 \# h$ for all h. Intuitively, $h_2 \# h$ still holds if h_1 and h_2 transfer inflows to outflows in the same way.

Formally, for a flow graph h we define its *transfer function* $tf(h)$ mapping inflows to outflows, $tf(h) : ((\mathbb{N} \setminus X) \times X \to \mathbb{M}) \to X \times (\mathbb{N} \setminus X) \to \mathbb{M}$, by

$$tf(h)(in') \triangleq h[in \mapsto in'].out\ .$$

For a given inflow in, we also write $tf(h_1) =_{in} tf(h_2)$ to mean that for all inflows $in' \leq in$, $tf(h_1)(in') = tf(h_2)(in')$.

Definition 1. *Flow graphs h_1, h_2 are* contextually equivalent*, denoted $h_1 =_{ctx} h_2$, if we have $h_1.X = h_2.X$, $h_1.in = h_2.in$, and $tf(h_1) =_{h_1.in} tf(h_2)$.*

Theorem 1 (Frame Preservation). *For all flow graphs $h_1 =_{ctx} h_2$ and h, $h_1 \# h$ if and only if $h_2 \# h$ and, in case of definedness, $h_1 * h =_{ctx} h_2 * h$.*

To automate the frame rule for a command c and a precondition P, we need to identify a decomposition $P = P_1 * F$ so as to infer $\{P_1\}\, c\, \{P_2\}$ and then apply the frame rule to derive $\{P\}\, c\, \{Q\}$ for the postcondition $Q = P_2 * F$. This is closely related to the *frame inference problem* [4]. When a command modifies a flow graph h_1 to h_2, our goal is to identify a (hopefully small) set of nodes Y in h_1 that are affected by this update, the *flow footprint*. That is, Y captures the difference between the flow graphs before and after the update and the complement of Y defines the frame. To make this formal, we need the restriction of flow graphs to subsets of nodes, which then gives us a notion of flow graph decomposition. Towards this, consider h and $Y \subseteq \mathbb{N}$. We define

$$h|_Y \ \triangleq\ (h.X \cap Y, h.E|_{(h.X \cap Y) \times \mathbb{N}}, in)$$

such that the inflow in satisfies $in(z, y) \triangleq h.in(z, y)$ for all $z \in \mathbb{N} \setminus h.X$, $y \in h.X \cap Y$ and $in(x, y) \triangleq h.E_{(x,y)}(h.flow(x))$ for all $x \in h.X \setminus Y$, $y \in h.X \cap Y$.

Definition 2. *Consider h_1 and h_2 with $X \triangleq h_1.X = h_2.X$ and $h_1.in = h_2.in$. A* flow footprint *for the difference between h_1 and h_2 is a subset of nodes $Y \subseteq X$ so that $h_1|_Y =_{ctx} h_2|_Y$ and $h_1|_{X \setminus Y} = h_2|_{X \setminus Y}$. The set of all such footprints is $FFP(h_1, h_2)$.*

Flow graphs over different sets of nodes or inflows never have a flow footprint. The former requirement merely simplifies the presentation. To that end, we assume that all nodes that will be allocated during program execution are already present in the initial flow graph. This assumption can be lifted. The latter requirement is motivated by the fact that the global inflow is part of the specification as noted earlier in Example 1.

Before we proceed with the problem of how to compute flow footprints, we highlight some of their properties.

Lemma 3 (Footprint Monotonicity). *If $Z \in FFP(h_1, h_2)$ and $Z \subseteq Y \subseteq h_1.X$, then $Y \in FFP(h_1, h_2)$.*

A consequence of monotonicity is the existence of a canonical flow footprint: if there is a flow footprint at all, then the set of all nodes will work as a footprint. Of course this canonical footprint is undesirably large. It corresponds to the case where one reasons about flow graph updates globally, forgoing the application of the frame rule. Unfortunately, an inclusion-minimal flow footprint does not exist.

Proposition 1 (Canonical Footprints). *We have: $FFP(h_1, h_2) \neq \varnothing$ if and only if $h_1.X \in FFP(h_1, h_2)$. There is no inclusion-minimal flow footprint; in particular, the set $FFP(h_1, h_2)$ is not closed under intersection.*

The proof of monotonicity requires a better understanding of the restriction operator, as provided by the following lemma.

Lemma 4 (Restriction). *Consider h and $Y, Z \subseteq \mathbb{N}$. Then (i) $h|_Y.flow = h.flow|_Y$, (ii) $h|_Y \# h|_{X \setminus Y}$ and $h|_Y * h|_{X \setminus Y} = h$, and (iii) $(h|_Y)|_Z = h|_{Y \cap Z}$.*

Since flow footprints are defined via restriction, the lemma also shows that flow footprints are well-behaved. For example, the restriction to the footprint Y does not change the flow of a node $y \in Y$ nor that of a node $x \in h.X \setminus Y$. More formally, this means $h|_Y.flow(y) = h.flow(y)$ and $h|_{X \setminus Y}.flow(x) = h.flow(x)$, by Lemma 4(i).

For our development, it will be convenient to have a more operational formulation of the transfer function. Towards this, we understand the flow graph as a function that takes an inflow as a parameter and yields a transformer of flow approximants:

$$h \;:\; ((\mathbb{N} \setminus X) \times X \to \mathbb{M}) \to (X \to \mathbb{M}) \to X \to \mathbb{M}$$

$$\text{defined by} \qquad h[in](\sigma)(x) \;=\; in_x + rhs_x(\sigma)\ .$$

Recall $in_x \triangleq \sum_{y \in \mathbb{N} \setminus X} in(y, x)$ and $rhs_x(\sigma) = \sum_{y \in X} E_{(y,x)}(\sigma(y))$. The least fixed point of $h[in]$ is $\bigsqcup_{i \in \mathbb{N}} h[in]^i(\bot)$ with $h^0 = id_{X \to \mathbb{M}}$ and $h^{i+1} = h^i \circ h$, by Kleene's theorem. Define $out : (X \to \mathbb{M}) \to X \times (\mathbb{N} \setminus X) \to \mathbb{M}$ by $out(\sigma)(y, z) \triangleq E_{(y,z)}(\sigma(y))$. This yields the following characterization of transfer functions and flows.

Lemma 5 (Transfer). *For all flow graphs h we have (i) $tf(h) = out \circ (lfp.h[-])$ and (ii) $lfp.h[h.in]) = h.flow$.*

4 Computing Footprints

We present an algorithm for computing a footprint for the difference between two given flow graphs. We proceed in two steps. We first give a high-level description of the algorithm that ignores computability problems. In a second step, we show how to solve the computability problems. Throughout the development, we will assume to have flow graphs h_1 and h_2 over the same nodes $X \triangleq h_1.X = h_2.X$ and with the same inflow $h_1.in = h_2.in$. If this assumption fails, a flow footprint does not exist by definition.

4.1 Algorithm

We compute the flow footprint as a fixed point. We start with the footprint candidate Z consisting of the nodes whose outgoing edges differ in h_1 and h_2. Then, we iteratively add the nodes whose outflow leaving the current footprint candidate Z differs in $h_1|_Z$ and $h_2|_Z$. That the outflow differs means that the transfer functions $tf(h_1|_Z)$ and $tf(h_2|_Z)$ differ and thus the candidate Z is not a footprint. In turn, if all outflows match, the transfer functions coincide and Z is a footprint as desired.

Technically, we compute the fixed point over the powerset lattice of nodes endowed with a distinguished top element: $(\mathbb{P}(X)^\top, \sqsubseteq)$ with $\mathbb{P}(X)^\top \triangleq \mathbb{P}(X) \uplus \{\top\}$. Element $\top$ indicates a failure of the footprint computation. This may arise if the footprint is not covered by X, i.e., extends beyond the flow graphs h_1, h_2.

Our fixed point computation starts from $Z = \mathit{odif}_{h_1,h_2} \subseteq X$ as defined by

$$\mathit{odif}_{h_1,h_2} \triangleq \{\, x \in X \mid \exists z \in \mathbb{N}. h_1.E(x,z) \neq h_2.E(x,z) \,\}\,.$$

The fixed point then proceeds to extend Z as long as the transfer functions associated with $h_1|_Z$ and $h_2|_Z$ do not match. To define the extension, we let the *transfer failure* of $Z \subseteq X$ be the successor nodes of Z that may receive different outflow from h_1 and h_2:

$$\mathit{tfail}_{h_1,h_2}(Z) \triangleq \left\{ x \in \mathbb{N} \setminus Z \;\middle|\; \begin{array}{l} \exists\, in \leq h_1|_Z.in \;\; \exists\, z \in Z. \\ \qquad [tf(h_1|_Z)(in)](z,x) \neq [tf(h_2|_Z)(in)](z,x) \end{array} \right\}.$$

This set is the *reason* why the current footprint candidate Z is not a footprint, that is, $Z \notin \mathit{FFP}(h_1, h_2)$. Extending Z with the transfer failure yields a new candidate. We check that the new candidate is covered by X (i.e., does not include nodes outside of h_1, h_2). If the check fails, the new candidate is $\{\top\}$ to indicate that no footprint could be computed. The following definition makes the extension procedure precise.

Definition 3. *The function* $\mathit{ext}_{h_1,h_2} : \mathbb{P}(X)^\top \to \mathbb{P}(X)^\top$ *is defined by*

$$\mathit{ext}_{h_1,h_2}(Z) \triangleq \mathit{tfail}_{h_1,h_2}(Z) \not\subseteq X \;\; \mathbf{?} \;\; \top \;\; \mathbf{:} \;\; Z \sqcup \mathit{odif}_{h_1,h_2} \sqcup \mathit{tfail}_{h_1,h_2}(Z)\,.$$

Iteratively extending the candidate Z with the transfer failure eventually produces a footprint for the difference of h_1 and h_2, or fails with $\top$. The approach is sound.

Theorem 2 (Soundness). *Let* $F \triangleq \mathit{lfp}.\mathit{ext}_{h_1,h_2}$. *If* $F \neq \top$, *then* $F \in \mathit{FFP}(h_1, h_2)$.

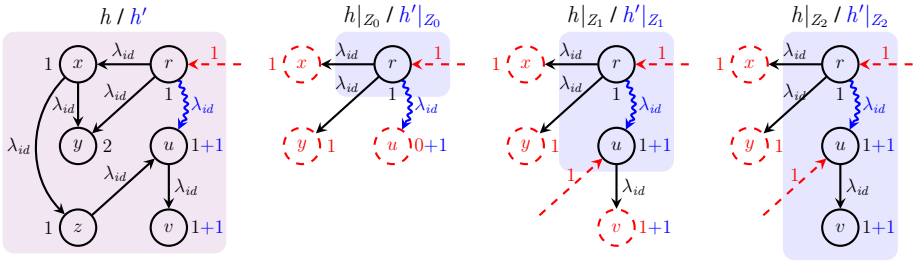


Figure 3. Computing a footprint for the difference of h and h' iterates through the sets $Z_0 \triangleq \{ r \}$, $Z_1 \triangleq \{ r, u \}$, and $Z_2 \triangleq \{ r, u, v \}$. The latter is the least fixed point of $ext_{h,h'}$ and a footprint as desired, $Z_2 \in FFP(h, h')$.

Example 2. For an illustration consider Fig. 3. There, we apply the fixed point computation to find a footprint for the difference of h and h'. As alluded to in Sect. 1, h' is the result of inserting into h a new edge between nodes r and u labeled with λ_{id}.

The fixed point computation starts from $Z_0 \triangleq \{ r \} = odif_{H,H'}$ as it is the only node whose outgoing edges have changed. Next, we compute $tfail_{h,h'}(Z_0)$. This yields $\{ u \}$ because u receives 0 from Z_0 in h but 1 in h' due to the new edge. The outflow from Z_0 to the remaining nodes coincides in h and h'. Hence, the extension of Z_0 with the transfer failure yields $Z_1 \triangleq ext_{h,h'}(Z_0) = \{ u, r \}$. Similarly, we compute $tfail_{h,h'}(Z_1)$ and obtain $Z_2 \triangleq ext_{h,h'}(Z_1) = \{ r, u, v \}$. Since v has no outgoing edges, Z_2 is the least fixed point of $ext_{h,h'}$. Because Z_2 is a subset of the nodes of h and h', it is a footprint, $Z_2 \in FFP(h, h')$. □

To obtain Theorem 2, we have to prove that the fixed point $F \triangleq lfp.ext_{h_1,h_2}$ is indeed a footprint if $F \neq \top$. That is, we have to establish the following two properties according to Definition 2: (i) $h_1|_F =_{ctx} h_2|_F$ and (ii) $h_1|_{X \setminus F} = h_2|_{X \setminus F}$.

To see the latter one, note that the graph structures (the nodes and edges) of $h_1|_{X \setminus F}$ and $h_2|_{X \setminus F}$ coincide because $odif_{h_1,h_2} \subseteq F$. The inflows coincide as well because they are, intuitively, comprised of the flow graph's overall inflow $h_1.in = h_2.in$ and the outflow of the footprint, which is equal in both flow graphs due to $h_1|_F =_{ctx} h_2|_F$.

The interesting part of the soundness proof is to establish property (i), the contextual equivalence $h_1|_F =_{ctx} h_2|_F$. Since F is a fixed point of ext_{h_1,h_2}, we know that $tfail_{h_1,h_2}(Z) = \varnothing$ and thus the transfer functions of $h_1|_F$ and $h_2|_F$ coincide. Hence, it suffices to establish $h_1|_F.in = h_2|_F.in$ to obtain the desired contextual equivalence, Definition 1. This key step in the proof is obtained with the help of the following lemma.

Lemma 6. *Let* $odif_{h_1,h_2} \subseteq F \subseteq X$ *with* $tfail_{h_1,h_2}(F) = \varnothing$. *Then* $h_1|_F.in = h_2|_F.in$.

To establish the lemma one has to show that the inflow into F from the non-footprint part $Y \triangleq X \setminus F$ coincides in h_1 and h_2. The challenge is a cyclic dependency in the flow: the inflow from Y depends on the outflow of F, which depends on the inflow from Y. To tackle this, we rephrase the flow equation for h_i as a pairing of the two separate flow equations for $h_i|_F$ and $h_i|_Y$, for $i \in \{ 1, 2 \}$. Intuitively, the pairings compute the flow locally in $h_i|_F$ and $h_i|_Y$ for a fixed inflow (initially $h_i.in$). Then, the inflow to $h_i|_F$

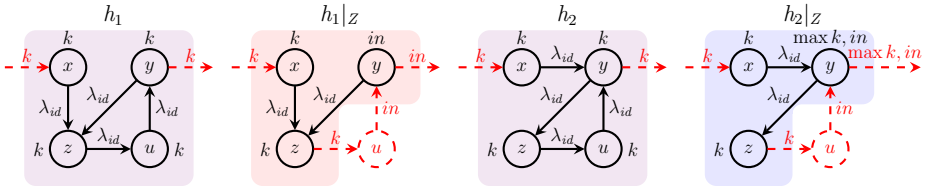


Figure 4. Counterexample to completeness using the monoid $(\mathbb{N}\cup\{\infty\}, \max, 0)$. While the set $\{x, y, z, u\}$ is a footprint for the difference between flow graphs h_1 and h_2, our fixed point will produce the candidates $\{x\}$ and $Z \triangleq \{x, y, z\}$ and then fail with $\{\top\}$.

is updated to the inflow from outside h_i and the inflow from $h_i|_Y$, and similarly for the inflow to $h_i|_Y$. This is repeated until a fixed point is reach. Technically, we rely on Bekić's Lemma [1] to compute the pairings. Then, we observe $tf(h_1|_F) = tf(h_2|_F)$ because $tfail_{h_1,h_2}(F) = \varnothing$ as well as $tf(h_1|_Y) = tf(h_2|_Y)$ because $odif_{h_1,h_2} \subseteq F$. Roughly, this means that the flow pairings for h_1 and h_2 must coincide as the individual parts propagate the same values. Put differently, the updated inflow for $h_1|_F$ and $h_2|_F$ as well as $h_1|_Y$ and $h_2|_Y$ coincide in each iteration. Overall, we get $h_1|_F.in = h_2|_F.in$.

Our computation of a flow footprint is forward, it starts from the nodes where the flow graphs differ and follows the edges. It may therefore fail if predecessor nodes of an iterate Z need to be considered to determine a flow footprint. For an example refer to Fig. 4. Using the monoid $(\mathbb{N}\cup\{\infty\}, \max, 0)$, it is easy to see that the set $\{x, y, z, u\}$ is a footprint for the difference between h_1 and h_2. Our fixed point, however, will start with $\{x\}$ and extend this to $Z \triangleq \{x, y, z\}$. Let v be the node outside the flow graphs that y is pointing to. Then, the next transfer failure is $tfail_{h_1,h_2}(Z) = \{v\}$ because for $in < k$ the outflow of y to v differs in $h_1|_Z$ and $h_2|_Z$. Our approach fails to compute a footprint.

Fact 3 (Incompleteness) *There are flow graphs h_1 and h_2 for which our algorithm is not able to determine a flow footprint although one exists.*

4.2 Comparing Transfer Functions

When implementing the above fixed point computation, the challenge is to prove the equivalence between given transfer functions in order to obtain the transfer failure: $[tf(h_1|_Z)(-)](-, x) = [tf(h_2|_Z)(-)](-, x)$? Already the comparison of two functions is known to be difficult to do algorithmically. What adds to the problem is that transfer functions are defined as least fixed points, meaning we do not have a closed-form representation of the functions to compare.

Our approach is to impose additional requirements on the set of edge functions. The requirements are met in all our experiments, and so do not mean a limitation for the applicability of our approach. We show that if the edge functions are not only continuous but also distributive, then the transfer functions can be understood in terms of paths through the underlying flow graphs. If the edge functions are additionally decreasing and the underlying monoid's addition is idempotent, then acyclic paths are sufficient. Both results do not hold for merely continuous edge functions.

Distributivity. Our first additional assumption is that the edge functions $f : \mathbb{M} \to \mathbb{M}$ are not only continuous, but also *distributive* in that $f(m + n) = f(m) + f(n)$ for all $m, n \in \mathbb{M}$ and $f(0) = 0$. We use $DistFun(\mathbb{M})$ to refer to the set of all continuous and distributive functions over $\mathbb{M}$. The properties formulated in Lemma 1 carry over.

For continuous and distributive transfer functions, we can understand $h[in]^i$ in terms of the paths through $h[in]$ of length i. For example, $i = 3$ yields

$$\begin{aligned}[h[in]^3](\bot)(z) &= in_z + \sum_{y \in X} E_{(y,z)}(\ in_y + \sum_{x \in X} E_{(x,y)}(in_x + \sum_{u \in X} E_{(u,x)}(\bot(u))\ )\\ &= in_z + \sum_{y \in X} E_{(y,z)}(in_y) + \sum_{y \in X} \sum_{x \in X} E_{(y,z)}(E_{(x,y)}(in_x))\ .\end{aligned}$$

The first equality is by definition, the second is where distributivity comes in. In particular, $\bot(u) = 0$ and so $E_{(y,z)}(\ E_{(x,y)}(\ E_{(u,x)}(\ \bot(u)\ )\ ) = 0$. The last term shows that we forward the inflow given at a node x to an intermediary node y and from there to the node z of interest. For higher powers of $h[in]$, we take longer paths. For $h[in]^*$, we thus obtain the sum over all nodes x and all paths from x to z through the flow graph. We need some definitions to make this precise.

A *path* p through flow graph h is a finite, non-empty sequence of nodes all of which belong to the flow graph except the last which lies outside:

$$p \ = \ x_0 \cdot \ldots \cdot x_n \cdot z \ \in \ X^+ \cdot (\mathbb{N} \setminus X)$$

where $\cdot$ denotes path concatenation. We use $first(p) = x_0$ resp. $last(p) = x_n$ to extract the first resp. last node from within the flow graph h. By $Paths(h, x, y, z)$ we denote the set of all paths through flow graph h that start in node $first(p) = x$ and leave h from node $last(p) = y$ to move to $z \in \mathbb{N} \setminus X$. Given a set of nodes $X' \subseteq X$, we use $Paths(h, X', y, z)$ for the union over all $x \in X'$ of the sets $Paths(h, x, y, z)$. The path induces the function $E_p : \mathbb{M} \to \mathbb{M}$ that composes the edge functions along the path:

$$E_x \ = \ id \qquad\qquad E_{x.p} \ = \ E_p \circ E_{(x,first(p))}\ .$$

Together with Lemma 5, the above analysis yields the first closed-form representation of a flow graph's transfer function, which so far has involved a fixed point computation.

Theorem 4 (Closed-Form Representation). *If h is labeled over $DistFun(\mathbb{M})$, then:*

$$[tf(h)(in)](y, z) \quad = \quad \textstyle\sum_{x \in X} \sum_{p \in Paths(h,x,y,z)} E_p(in_x)\ .$$

Theorem 4 pushes the fixed point computation of transfer functions into the sets $Paths(h, x, y, z)$ which are themselves defined inductively and potentially infinite. In the following, we alleviate this problem without requiring acyclicity of the flow graph.

Idempotence. Our second assumption is that addition in the monoid is idempotent, meaning $m + m = m$ for all $m \in \mathbb{M}$. Idempotence ensures the addition degenerates to a join for comparable elements: $m + n = m \sqcup n = n$ for all $m \leq n \in \mathbb{M}$. Unless stated otherwise, we hereafter assume an idempotent addition.

With Theorem 4, it remains to compare sums over paths. With idempotence, we show that we can further reduce the problem and reason over single paths rather than

sums. We show that every path in h_1 can be replaced by a set of paths in h_2, and vice versa. Even more, we only have to consider the paths from nodes where the edges changed. The precise formulation of the path replacement condition is the following.

Definition 4. *The* path replacement condition *for flow graphs h_1 by h_2 over the same set of nodes X and labeled by $DistDecFun(\mathbb{M})$ requires that for every $x \in odif_{h_1,h_2}$, for every $y \in X$, and for every $z \in \mathbb{N} \setminus X$ we have*

$$\forall\, p \in Paths(h_1, x, y, z)\ \exists\, P \subseteq Paths(h_2, x, y, z).\quad E_p \ \leq\ E_P \ \triangleq\ \textstyle\sum_{q \in P} E_q\ .$$

Example 3. For the flow graphs h_1 and h_2 from Fig. 4, we have path replacement of h_1 by h_2, and vice versa. To see this, consider the path $p \triangleq x \cdot z \cdot u \cdot y \cdot v$ in h_1 and $q \triangleq x \cdot y \cdot v$ in h_2, where v is the node outside of h_1, h_2 that y points to. Since all edges are labeled with λ_{id}, we have $E_p = \lambda_{id} = E_q$. It is worth noting that, in this example, we can ignore the cycles in h_1 and h_2. In a moment, we will introduce restrictions on edge functions in order to do avoid cycles in general.

Similarly, we have path replacement for the flow graphs from Fig. 2. To be precise, $E_p = \lambda_8 = E_q$ for the paths $p \triangleq l \cdot t \cdot r \cdot v$ in h_1 and $q \triangleq l \cdot r \cdot v$ in h_2. □

The main result is that path replacement is sound and complete for proving equivalence of transfer functions.

Theorem 5 (Path Replacement Principle). *We have $tf(h_1) = tf(h_2)$ if and only if path replacement of h_1 by h_2 and of h_2 by h_1 hold.*

The theorem is remarkable in several respects. First, one would expect we have to replace the paths from all nodes in h_1. Instead, we can focus on the nodes where the outgoing edges changed. Second, one would expect the replacing paths P start from arbitrary nodes in h_2. Such a set of paths would yield a transfer function of type $(Y \to \mathbb{M}) \to \mathbb{M}$. Instead, we can work with a function of type $\mathbb{M} \to \mathbb{M}$. Even more, we can focus on paths starting in the same node as the path we intend to replace. Finally, the paths we use for replacement come without any constraints, leaving room for heuristics.

The proof starts from a *full path replacement condition* of h_1 by h_2, both over X and labeled by $DistFun(\mathbb{M})$. Full path replacement coincides with Definition 4 but draws x from full X rather than $x \in odif_{h_1,h_2}$. Full path replacement characterizes equivalence of the transfer functions in a monoid with idempotent addition in the case of continuous and distributive edge functions.

Lemma 7. *Full path replacement of h_1 by h_2 and h_2 by h_1 hold iff $tf(h_1) = tf(h_2)$.*

The result is a consequence of Theorem 4, which equates $tf(h_1)$ with the sum of the E_p for all paths $p \in Paths(h_1, x, y, z)$ for all $x \in X$. Full path replacement allows us to sum over E_P instead, for some $P \subseteq Paths(h_2, x, y, z)$. Over-approximating P with all paths $Paths(h_2, x, y, z)$, we obtain an upper bound for $tf(h_1)$. It is easy to see that the resulting sum can be rewritten into the form of Theorem 4, yielding $tf(h_1) \leq tf(h_2)$. Analogously, we get $tf(h_1) \geq tf(h_2)$ and thus $tf(h_1) = tf(h_2)$ as required. The reverse direction of the lemma is similar.

To conclude the proof of the path replacement principle in Theorem 5, we show that full path replacement and (ordinary) path replacement of h_1 by h_2 coincide. To see this,

consider a path $p \in Paths(h_1, x, y, z)$ for any $x \in X$. The goal is to show $E_p \leq E_P$ for some $P \in Paths(h_2, x, y, z)$. To that end, decompose the path into $p = p_1 \cdot p_2$ such that $x' \triangleq first(p_2)$ is the first node in p from $odif_{h_1,h_2}$. Ordinary path replacement yields $Q \in Paths(h_2, x', y, z)$ with $E_{p_2} \leq E_Q$. Now, choose $P \triangleq \{\, p_1 \cdot q \mid q \in Q \,\}$. Because p_1 exists in h_1 and h_2 with the exact same edge labels, we obtain the desired $E_p \leq E_P$.

Lemma 8. *Full path replacement of h_1 by h_2 holds if and only if path replacement of h_1 by h_2 holds.*

Decreasingness. We assume that the edge functions $f : \mathbb{M} \to \mathbb{M}$ are not only continuous and distributive, but also *decreasing*: $f(m) \leq m$ for all $m \in \mathbb{M}$. The assumption of decreasing edge functions is justified by the fact that a program that traverses the flow graph builds up information about the status of the structure, and smaller flow values mean more information (as in classical data flow analysis). We use $DistDecFun(\mathbb{M})$ to refer to the set of all continuous, distributive, and decreasing transfer functions over $\mathbb{M}$; Lemma 1 carries over to this set. Addition in the monoid is still assumed idempotent.

If all edge functions are decreasing, every cycle in the flow graph is decreasing as well. The key observation is that, given an idempotent addition, cycles with decreasing edge functions can be avoided when forming sums over sets of paths.

Lemma 9. *Let h be labeled over $DistDecFun(\mathbb{M})$ and $p_1 \cdot p \cdot p_2 \in Paths(h, x, y, z)$ with $last(p) = first(p)$. Then $p_1 \cdot p_2 \in Paths(h, x, y, z)$ and $E_{p_1 \cdot p \cdot p_2} \leq E_{p_1 \cdot p_2}$.*

Call a path *simple* if it does not repeat a node and let $SimplePaths(h, x, y, z)$ denote the set of all simple paths through h from x to y and leaving the flow graph towards z. Note that a finite graph only admits finitely many simple paths.

Theorem 6 (Simple Paths). *Assuming continuous, distributive, and decreasing edge functions, and assuming idempotent addition, Theorem 4 and Theorem 5 hold with every occurrency of $Paths(h, x, y, z)$ replaced by $SimplePaths(h, x, y, z)$.*

In practice, path-counting flows, keyset flows, reachability flows, shortest-path flows, and priority inheritance flows are relevant [22–24, 27] and compatible with our theory.

5 Evaluation

We substantiate the practicality of our new approach by evaluating it on a real-world collection of flow graphs extracted from the literature. We explain how we obtained our benchmarks and how we implemented and evaluated our approach.

Benchmark Suite. As alluded to in Sect. 1, the flow framework has been used to verify complex concurrent data structures. More specifically, it has been used for automated proof construction by the `plankton` tool [26, 27]. `plankton` performs an exhaustive proof search over a separation logic with support for flows—and further advanced features for establishing linearizability that do not matter for the present evaluation. In order to handle heap updates, `plankton` generates a footprint h for the flow graph $h_1 = h * h_{frame}$ of the current proof state (represented as an assertion in separation

logic). It then frames the non-footprint part h_{frame} of the flow graph h_1 to compute the post state h' of the heap update locally for the footprint h. The result is the new flow graph $h_2 = h' * h_{frame}$. We consider the pair (h_1, h_2) a *benchmark* for our evaluation.

We adapt `plankton` to export the flow graph pairs for which a footprint is constructed. This way, we obtain 1272 benchmarks from the heap updates occurring during proof construction for a collection of 10 concurrent set data structures. All flow graphs in this benchmark suite contain at most 4 nodes.

Our benchmark suite is limited by the capabilities and restrictions of `plankton`. In particular, we inherit the confinement to concurrent search structures. This is due to the fact that `plankton` integrates support only for the keyset flow (cf. Example 1). Our evaluation will compute footprints with respect to this flow.

Implementation. We implement the fixed point computation to find footprints for two given flow graphs h_1, h_2 from Sect. 4 in a tool called `krill` [28]. It integrates three methods for computing the transfer failure $tfail_{h_1,h_2}(Z)$ of a footprint candidate Z:

1. **NAIVE:** A naive method that computes the flow within the footprint Z. Following [24], we require acyclicity of flow graphs for this method to avoid solving a fixed point equation when computing the flow.
2. **NEW:** Our new approach leveraging the path replacement condition (cf. Theorem 5) for simple paths (cf. Theorem 6). This method requires distributive and decreasing edge functions as well as idempotent addition in the underlying monoid.
3. **DIST:** A variation of our new approach leveraging the closed-form representation (cf. Theorem 4). We require distributive edge functions and acyclicity of the flow graphs to avoid an unbounded sum over all paths in the closed-form representation.

Our benchmark suite satisfies the requirements for all three methods. The NAIVE and DIST methods include a (sufficient) check to ensure acyclicity in the updated flow graph to guarantee soundness of the resulting footprint.

All three methods encode the necessary equivalence checks among transfer functions as SMT formulas which are then discharged using the off-the-shelf SMT solver `Z3` [31]. Our encodings use the theory of integers with quantifiers. The NAIVE method additionally uses free functions to encode sets of integers.

Experiments. We ran `krill` on our benchmark suite and compared the runtime of the three different methods for computing the transfer failure. Our results are summarized in Fig. 5(left). For every search structure that we extracted benchmarks from, the figure lists: (i) the number #FG of flow graph pairs extracted, (ii) each method's total runtime for computing the footprints of all flow graph pairs, and (iii) the speedup of NEW over NAIVE in percent. The experiments were conducted on an Apple M1 Pro.

Figure 5(left) shows that the runtime for all methods is roughly linear in the number of computed footprints. Moreover, the absolute time for computing footprints is small, making the approaches practical. The figure also shows that our NEW and DIST methods have a performance advantage over the NAIVE method. The NEW method is between 22% and 39% faster than the NAIVE method. We believe that the difference is relatively small only because the acyclicity assumption avoids a potentially non-terminating fixed point computation. Avoiding this fixed point in the presence of cycles is a major advantage that our NEW method has over the NAIVE and DIST methods. The performance difference for DIST and NEW are negligible because the acyclicity check is negligible.

Structure	#FG	NAIVE	DIST	NEW	Speedup
Fine set [13]	12	75 ms	48 ms	46 ms	39%
Lazy set [12]	14	73 ms	52 ms	51 ms	30%
ORVYY set [33]	20	106 ms	76 ms	74 ms	30%
VY DCAS set [46]	19	109 ms	74 ms	73 ms	33%
VY CAS set [46]	28	139 ms	104 ms	102 ms	27%
Michael set [29]	225	1216 ms	887 ms	874 ms	28%
Michael set (wait-free)	186	996 ms	731 ms	721 ms	27%
Harris set [11]	352	2242 ms	1490 ms	1443 ms	36%
Harris set (wait-free)	296	1859 ms	1242 ms	1205 ms	35%
FEMRS tree [10]	120	519 ms	409 ms	407 ms	22%
Total	1272	7335 ms	5114 ms	4996 ms	32%

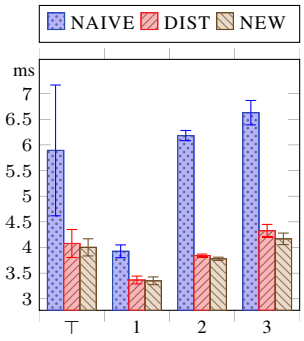


Figure 5. Experimental results averaged over 1000 repeated runs, conducted on an Apple M1 Pro. **(left)** Total runtime for computing footprints for flow graphs occurring during automated proof construction for highly concurrent set data structures. The speedup gives the relative performance improvement of NEW over NAIVE. **(right)** Average runtime for computing a single footprint, partitioned by footprint size ($\top$ indicates failure).

We also factorized the runtimes of our benchmarks along the size of the resulting footprint. Figure 5(right) gives the average runtime and standard deviation for computing a single footprint, broken down by footprint size. If no footprint could be found, its size is listed as $\top$. These failed footprint constructions are consistent with `plankton`'s method and would not lead to verification failure.

6 Related Work

Two alternative meta theories for the flow framework have been proposed in prior work [23, 24]. Like in our setup, the original flow framework [23] demands that the flow domain is an ω-cpo to obtain a least fixed point semantics. However, it proposes a different flow graph composition that leads to a notion of contextual equivalence relying on inflow equivalence classes. This complicates proof automation. In addition, the flow domain is assumed to be a semiring and edge functions are restricted to multiplication with a constant. This limits expressivity.

As discussed in Sect. 1, the revised flow framework proposed in [24] requires that the flow monoid is cancellative but not an ω-cpo. This means that uniqueness of flows is not guaranteed per se. Instead, uniqueness is obtained by imposing additional conditions on the edge functions. However, these conditions are more restrictive than those imposed in our framework. The *capacity* of a flow graph introduced in [24] closely relates to our notion of transfer function. A closed-form representation based on sums over paths is used to check equivalence of capacities. However, this reasoning is restricted to acyclic graphs. Also, [24] provides no algorithm for computing flow footprints.

In a sense, our work strikes a balance between the two prior meta theories by guaranteeing unique flows without sacrificing expressivity and, at the same time, enabling better proof automation. That said, we believe that the framework proposed in [24] remains of independent interest, in particular if the application does not require unique

flows (i.e., does not impose lower bounds on flows that may trivially hold in the presence of vanishing flows). Cancellativity allows one to aggregate inflows and outflows to unary functions, which can lead to smaller flow footprints (i.e., more local proofs).

The benchmark suite for our evaluation is obtained from `plankton` [26,27], a tool for verifying concurrent search structures using keyset flows. When the program mutates the symbolic heap, `plankton` creates a flow graph for the mutated nodes plus all nodes with a distance of k or less from those nodes. This flow graph is considered to be the footprint and contextual equivalence is checked. The check is basically the same as for NAIVE. However, the paper does not present the meta theory for the underlying notion of flow graphs, nor does it provide any justification for the correctness of the implemented algorithms used to reason about flow graphs.

Flow graphs form a separation algebra. Hence, the developed theory can be used in combination with any existing separation logic that is parametric in the underlying separation algebra such as [5, 7, 18, 27, 41, 44]. Identifying footprints of updates relates to the frame inference problem in separation logic, which has been studied extensively [4, 6, 15, 25, 35, 36, 42]. However, existing work focuses on frame inference for assertions that are expressed in terms of inductive predicates. These techniques are not well-suited for reasoning about programs manipulating general graphs, including overlayed structures, which are often used in practice and easily expressed using flows. A common approach to reason about general heap graphs in separation logic is to use iterated separating conjunction [14, 39, 44, 47] to abstract the heap by a *pure* graph that does not depend on the program state. Though, the verification of specifications that rely on inductive properties of the pure graph then resorts back to classical first-order reasoning and is difficult to automate. An exception is [45] which uses SMT solvers to frame binary reachability relations in graphs that are described by iterated separating conjunctions. However, the technique is restricted to such reachability properties only.

Unbounded footprints have been encountered early on when computing the post image for recursive predicates [8]. This has spawned interest in separation logic fragments for which the reasoning can be efficiently automated [2,3,9,17,20,35,38]. A limitation that underlies all these works is an assumption of tree-regularity of the heap, in one way or another, which flows have been designed to overcome. In cases where the program (or ghost code) traverses the unbounded footprint (before or after the update), recent works [24, 27] have found a way to reduce the reasoning to bounded footprint chunks.

The definition of a flow closely resembles the classical formulation of a forward data flow analysis. The fact that the least fixed point of the flow equation for distributive edge functions can be characterized as a join over all paths in the flow graph mirrors dual results for greatest fixed points in data flow analysis [19,21]. In a similar vein, the notion of contextual equivalence of flow graphs relates to contextual program equivalence and fully abstract models in denotational semantics [16,30,37]. In fact, Bekić's Lemma [1], which we use in the proofs of Theorem 1 and lemma 6, was originally motivated by the study of such models. Flow graphs can serve as abstractions of programs (rather than just program states). We therefore believe that our results could also be of interest for developing incremental and compositional data flow analysis frameworks.

Data Availability Statement

The `krill` artifact and dataset generated and/or analysed in the present paper are available in the Zenodo repository [28], https://zenodo.org/record/7566204.

Acknowledgments

This work is funded in part by NSF grant 1815633. The first author was supported by the DFG project *EDS@SYN: Effective Denotational Semantics for Synthesis*. The third author is supported by a Junior Fellowship from the Simons Foundation (855328, SW).

References

1. Bekić, H.: Definable operation in general algebras, and the theory of automata and flowcharts. In: Programming Languages and Their Definition. Lecture Notes in Computer Science, vol. 177, pp. 30–55. Springer (1984)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: FSTTCS. Lecture Notes in Computer Science, vol. 3328, pp. 97–109. Springer (2004)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 115–137. Springer (2005)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: APLAS. Lecture Notes in Computer Science, vol. 3780, pp. 52–68. Springer (2005)
5. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS. pp. 366–378. IEEE (2007)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL. pp. 289–300. ACM (2009)
7. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300. ACM (2013)
8. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS. Lecture Notes in Computer Science, vol. 3920, pp. 287–302. Springer (2006)
9. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: SPEN: A solver for separation logic. In: NFM. Lecture Notes in Computer Science, vol. 10227, pp. 302–309 (2017)
10. Feldman, Y.M.Y., Enea, C., Morrison, A., Rinetzky, N., Shoham, S.: Order out of chaos: Proving linearizability using local views. In: DISC. LIPIcs, vol. 121, pp. 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
11. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. Lecture Notes in Computer Science, vol. 2180, pp. 300–314. Springer (2001)
12. Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS. Lecture Notes in Computer Science, vol. 3974, pp. 3–16. Springer (2005)
13. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
14. Hobor, A., Villard, J.: The ramifications of sharing in data structures. In: POPL. pp. 523–536. ACM (2013)
15. Holík, L., Peringer, P., Rogalewicz, A., Soková, V., Vojnar, T., Zuleger, F.: Low-level bi-abduction. In: ECOOP. LIPIcs, vol. 222, pp. 19:1–19:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
16. Hyland, J.M.E., Ong, C.L.: On full abstraction for PCF: i, ii, and III. Inf. Comput. **163**(2), 285–408 (2000)

17. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. In: ATVA. Lecture Notes in Computer Science, vol. 8837, pp. 201–218. Springer (2014)
18. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018)
19. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica **7**, 305–317 (1977)
20. Katelaan, J., Zuleger, F.: Beyond symbolic heaps: Deciding separation logic with inductive definitions. In: LPAR. EPiC Series in Computing, vol. 73, pp. 390–408. EasyChair (2020)
21. Kildall, G.A.: A unified approach to global program optimization. In: POPL. pp. 194–206. ACM Press (1973)
22. Krishna, S., Patel, N., Shasha, D.E., Wies, T.: Verifying concurrent search structure templates. In: PLDI. pp. 181–196. ACM (2020)
23. Krishna, S., Shasha, D.E., Wies, T.: Go with the flow: compositional abstractions for concurrent data structures. Proc. ACM Program. Lang. **2**(POPL), 37:1–37:31 (2018)
24. Krishna, S., Summers, A.J., Wies, T.: Local reasoning for global graph properties. In: ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 308–335. Springer (2020)
25. Le, Q.L., Sun, J., Qin, S.: Frame inference for inductive entailment proofs in separation logic. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 41–60. Springer (2018)
26. Meyer, R., Wies, T., Wolff, S.: Artifact for "A Concurrent Program Logic with a Future and History" (Sep 2022). https://doi.org/10.5281/zenodo.7080459
27. Meyer, R., Wies, T., Wolff, S.: A concurrent program logic with a future and history. Proc. ACM Program. Lang. **6**(OOPSLA) (2022)
28. Meyer, R., Wies, T., Wolff, S.: Artifact for "Make flows small again: revisiting the flow framework" (Jan 2023). https://doi.org/10.5281/zenodo.7566204
29. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. pp. 73–82. ACM (2002)
30. Milner, R.: Fully abstract models of typed *lambda*-calculi. Theor. Comput. Sci. **4**(1), 1–22 (1977)
31. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
32. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (2001)
33. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC. pp. 85–94. ACM (2010)
34. Patel, N., Krishna, S., Shasha, D.E., Wies, T.: Verifying concurrent multicopy search structures. Proc. ACM Program. Lang. **5**(OOPSLA), 1–32 (2021)
35. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 773–789. Springer (2013)
36. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 711–728. Springer (2014)
37. Plotkin, G.D.: LCF considered as a programming language. Theor. Comput. Sci. **5**(3), 223–255 (1977)
38. Qiu, X., Wang, Y.: A decidable logic for tree data-structures with measurements. In: VMCAI. Lecture Notes in Computer Science, vol. 11388, pp. 318–341. Springer (2019)
39. Raad, A., Hobor, A., Villard, J., Gardner, P.: Verifying concurrent graph algorithms. In: APLAS. Lecture Notes in Computer Science, vol. 10017, pp. 314–334 (2016)
40. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)

41. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Tada: A logic for time and data abstraction. In: ECOOP. Lecture Notes in Computer Science, vol. 8586, pp. 207–231. Springer (2014)
42. Rowe, R.N.S., Brotherston, J.: Automatic cyclic termination proofs for recursive procedures in separation logic. In: CPP. pp. 53–65. ACM (2017)
43. Scott, D.: Outline of a mathematical theory of computation. Tech. Rep. PRG02, Oxford University Computing Laboratory (1970)
44. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: PLDI. pp. 77–87. ACM (2015)
45. Ter-Gabrielyan, A., Summers, A.J., Müller, P.: Modular verification of heap reachability properties in separation logic. Proc. ACM Program. Lang. **3**(OOPSLA), 121:1–121:28 (2019)
46. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI. pp. 125–135. ACM (2008)
47. Yang, H.: An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In: Proceedings of the SPACE Workshop (2001)

ALASCA: Reasoning in Quantified Linear Arithmetic

Konstantin Korovin[3], Laura Kovács[1], Giles Reger[3], Johannes Schoisswohl[1](✉), and Andrei Voronkov[2,3]

[1] TU Wien, Vienna, Austria
johannes.schoisswohl@tuwien.ac.at
[2] EasyChair, Manchester, UK
[3] University of Manchester, Manchester, UK

Abstract. Automated reasoning is routinely used in the rigorous construction and analysis of complex systems. Among different theories, arithmetic stands out as one of the most frequently used and at the same time one of the most challenging in the presence of quantifiers and uninterpreted function symbols. First-order theorem provers perform very well on quantified problems due to the efficient superposition calculus, but support for arithmetic reasoning is limited to heuristic axioms. In this paper, we introduce the Alasca calculus that lifts superposition reasoning to the linear arithmetic domain. We show that Alasca is both sound and complete with respect to an axiomatisation of linear arithmetic. We implemented and evaluated Alasca using the Vampire theorem prover, solving many more challenging problems compared to state-of-the-art reasoners.

Keywords: Automated Reasoning · Linear Arithmetic · SMT · Quantified First-Order Logic · Theorem Proving

1 Introduction

Automated reasoning is undergoing a rapid development thanks to its successful use, for example, in mathematical theory formalisation [15], formal verification [16] and web security [13]. The use of automated reasoning in these areas is mostly driven by the application of SMT solving for quantifier-free formulas [6, 12, 29]. However, there exist many use case scenarios, such as expressing arithmetic operations over memory allocation and financial transactions [1, 18, 20, 32], which require complex first-order quantification. SMT solvers handle quantifiers using heuristic instantiation in domain-specific model construction [10, 28, 30, 36]. While being incomplete in most cases, instantiation requires instances to be produced to perform reasoning, which can lead to an explosion in work required for quantifier-heavy problems. What is rather needed to address the above use cases is a reasoning approach able to handle both theories and complex applications of quantifiers. Our work tackles this challenge and designs a *practical, low-cost methodology* for proving first-order quantified linear arithmetic properties.

S. Sankaranarayanan and N. Sharygina (Eds.) TACAS 2023, LNCS 13993, pp. 647–665, 2023.
https://doi.org/10.1007/978-3-031-30823-9_33

The problem of combining quantifiers with theories, and especially with arithmetic, is recognised as a major challenge in both SMT and first-order proving communities. In this paper *we focus on first-order, i.e. quantified, reasoning with linear arithmetic and uninterpreted functions.* In [26], it is shown that the validity problem for first-order reasoning with linear arithmetic and uninterpreted functions is Π^1_1-complete even when quantifiers are restricted to non-theory sorts. Therefore, there is no sound and complete calculus for this logic.

Quantified Reasoning in Linear Arithmetic – Related Works. In practice, there are two classes of methods of reasoning in first-order theory reasoning, and in particular with linear real arithmetic. SMT solvers use *instance-based methods*, where they repeatedly generate ground, that is quantifier-free, instances of quantified formulas and use decision procedures to check satisfiability of the resulting set of ground formulas [10, 28, 36]. Superposition-based first-order theorem provers use *saturation algorithms* [14, 27, 37]. In essense, they start with an initial set of clauses obtained by preprocessing the input formulas (initial search space) and repeatedly apply inference rules (such as superposition) to clauses in the search space, adding their (generally, non-ground) consequences to the search space. These two classes of methods are very different in nature and complement each other.

The superposition calculus [4, 31] is a refutationally complete calculus for first-order logic with equality that is used by modern first-order provers, for example, Vampire [27], E [37], iProver [17] and Zipperposition [14]. There have been a number of practical extensions to this calculus for reasoning in first-order theories, in particular for linear arithmetic [9, 11, 24]. Superposition theorem provers have become efficient and powerful on theory reasoning after the introduction of the AVATAR architecture [33, 38], which allows generated ground clauses to be passed to SMT solvers. Yet, superposition theorem provers have a major source of inefficiency. To work with theories, one has to add *theory axioms*, for example the transitivity of inequality $\forall x \forall y \forall z (x \leq y \wedge y \leq z \rightarrow x \leq z)$. In clausal form, this formula becomes $\neg x \leq y \vee \neg y \leq z \vee x \leq z$ where $\neg x \leq y$ can be resolved against *every* clause in which an inequality literal $s \leq t$ is selected. This, with other prolific theory axioms, results in a very significant growth of the search space. Note that SMT solvers do not use and do not need such theory axioms.

A natural solution is to try to eliminate some theory axioms, but this is notoriously difficult both in theory and in practice. In [26], the Lasca calculus was proposed, which replaced several theory axioms of linear arithmetic, including transitivity of inequality, by a new inference rule inspired by Fourier-Motzkin elimination and some additional rules. Lasca was shown to be complete for the ground case. But, after 15 years, Lasca is still not implemented, due to its complexity and lack of clear treatment for the non-ground case. As we argue in Sect. 5, lifting Lasca to the non-ground setting is nearly impossible as a non-ground extension of the underlining ordering is missing in [26].

Lifting Lasca to Alasca– Our contributions. In this paper we introduce a new non-ground version of Lasca, which we call Abstracting Lasca (Alasca). Our Alasca calculus comes with new abstraction mechanisms (Sect. 4), inference

rules and orderings (Sect. 5), which all together are proved to yield a sound and complete approach with respect to a natural partial axiomatisation of linear arithmetic (Theorem 5)[4]. In a nutshell, we make Alasca both work and scale by introducing (i) a novel variable elimination rule within saturation-based proof search (Fig. 3b); (ii) an analogue of *unification with abstraction* [34] needed for non-ground reasoning (Sect. 4); and (iii) a new non-ground ordering and powerful background theory for unification, which is not restricted to arithmetic but can be used with arbitrary theories (Sect. 5). As a result, Alasca improves [26] by ground modifications and lifting of Lasca in a finitary way, and complements [3, 40] with variable elimination rules that are competible with standard saturation algorithms. We also *demonstrate the practicality and efficiency* of Alasca (Sect. 6). To this end, we implemented Alasca in Vampire and show that it solves overall more problems than existing theorem provers.

2 Motivating Example

Consider the following mathematical property:

$$\forall x, y.\big(f(2x, y) > 2x + y \vee f(x + 1, y) > x + 2y\big) \rightarrow \forall x.\exists y.f(2, y) > x \qquad (1)$$

where f is an uninterpreted function. While property (1) holds, deriving its validity is hard for state-of-the-art reasoners: only veriT [2] can solve it. Despite its seeming simplicity, this problem requires non-trivial handling of quantifiers and arithmetic. Namely, one would need to unify (modulo theory) the terms $2x$ and $x + 1$ (which can be done by instantiating x with 1) and then derive $f(2, y) > 2 + y \vee f(2, y) > 1 + 2y$. Further, one also needs to prove that $f(2, y)$ is always greater than the minimum of $2 + y$ and $1 + 2y$, for arbitrary y.

Vampire with Alasca finds a remarkably short proof as shown in Fig. 1. To prove (1) its negation is shown unsatisfiable by first negating and translating into clausal form (by using skolemization and normalisation, which shifts arithmetic terms to be compared to 0), as listed in lines 1–4. Next a lower bound for $f(2x, y)$ is established: In line 5, using our new inequality factoring (IF) rule with unification with abstraction (see Fig. 3a), the constraint $2x \not\approx x + 1$ is introduced, and establishing thereby that if $2x \approx 1 + x$ and $y + 2x \leq 2y + x$, then $f(2x, y) > 2x + y$. After further normalisation, the inequalities $sk \geq f(2, y)$ and $f(2x, y) > 2x + y$ are used to derive $sk > 2x + y$ in line 7, using the Fourier-Motzkin Elimination rule (FM), while still keeping track of the constraint $2x \not\approx x + 1$. By applying the Variable Elimination rule (VE) twice, the empty clause $\square$ is derived in line 10, showing the unsatisfiability of the negation of (1).

The key steps in the proof (and the reason why it was found in a short time) are: (1) the use of the theory rules (FM), and (IF); (2) the use of the new variable elimination rule (VE), and finally, a consistent use of unification with abstraction. These rules give a significant reduction compared to the number of steps required using theory axioms. In particular, not using (FM) would require the use of transitivity and generation of several intermediate clauses. As well as shortening

[4] proofs and further details of our results can be found in [23]

1.	$f(2x,y) > 2x + y \vee f(x+1,y) > x + 2y$	Hypothesis
2.	$\neg f(2,y) > sk$	Skolemized, Neg. Conj.
3.	$f(2x,y) - 2x - y > 0 \vee f(x+1,y) - x - 2y > 0$	Normalisation 1
4.	$-f(2,y) + sk \geq 0$	Normalisation 2
5.	$f(2x,y) - 2x - y > 0 \vee y + 2x - 2y - x > 0 \vee 2x \not\approx x + 1$	(IF) 3
6.	$f(2x,y) - 2x - y > 0 \vee x - y > 0 \vee 0 \not\approx x - 1$	Normalisation 5
7.	$-2x - y + sk > 0 \vee x - y > 0 \vee 0 \not\approx x - 1 \vee 2x \not\approx 2$	(FM) 6,4
8.	$-2x - y + sk > 0 \vee x - y > 0 \vee 0 \not\approx x - 1$	Normalisation 7
9.	$0 \not\approx x - 1$	(VE) 8
10.	$\square$	(VE) 9

Fig. 1. A refutational proof using the calculus introduced in this paper. Variables x, y are implicitly universally quantified, and sk is an uninterpreted constant.

the proof, we eliminate the fatal impact on proof search from generating a large number of irrellevant formulas from theory axioms.

Indeed, such short proofs are also found quickly. Similar our previous example, $\forall x, y.\big(f(g(x)+g(a), y) > 2x+y \vee f(2g(x), y) > x+2y\big) \rightarrow \exists k. \forall x \exists z. f(2g(k), z) > x$ has a short proof of 7 steps, excluding CNF transformation and normalisation steps, found by Vampire with Alasca. This proof was found in almost no time (only 37 clauses were generated) but cannot be solved by any other solver. This shows the power of the calculus.

3 Background and Notation

Multi-Sorted First-Order Logic. We assume familiarity with standard first-order logic with equality, with all standard boolean connectives and quantifiers in the language. We consider a multi-sorted first-order language, with sorts $\tau_{\mathbb{Q}}, \tau_1, \ldots, \tau_n$. The sort $\tau_{\mathbb{Q}}$ is the *sort of rationals*, whereas $\tau_1, \ldots, \tau_n$ are *uninterpreted sorts*. We write $\approx_\tau$ for the equality predicate of τ. We denote the set of all terms as $\mathbf{T}$, variables as $\mathbf{V}$, and literals as $\mathbf{L}$. Throughout this paper, we denote terms by s, t, u, variables by x, y, z, function symbols by f, g, h, all possibly with indices. Given a term t such that t is $f(\ldots)$, we write $\mathsf{sym}(t)$ for f, referring that f is the top level symbol of t. We write $t : \tau$ to denote that t is a term of sort τ. A term, or literal is called *ground*, when it does not contain any variables. We refer to the sets of all ground terms, and literals as $\mathbf{T}^\theta$, and $\mathbf{L}^\theta$ respectively.

We denote predicates by P, Q, literals by L, clauses by C, D, formulas by F, G, and sets of formulas (axioms) by $\mathcal{E}$, possibly with indices. We write $F \models G$ to denote that whenever F holds in a model, then G does as well. We call a function (similarly, for predicates) f uninterpreted wrt some set of equations $\mathcal{E}$ if whenever $\mathcal{E} \models f(s_1 \ldots s_n) \approx f(t_1 \ldots t_n)$, then $\mathcal{E} \models s_1 \approx t_1 \wedge \ldots \wedge s_n \approx t_n$. A function f is interpreted wrt $\mathcal{E}$ if it is not uninterpreted.

Rational Sort. We assume the signature contains a countable set of unary functions $k : \tau_{\mathbb{Q}} \mapsto \tau_{\mathbb{Q}}$ for every $k \in \mathbb{Q}$ and refer to k as *numeral multiplications*. In addition, the signature is assumed to also contain a constant $1 : \tau_{\mathbb{Q}}$, a function

$+ : \tau_{\mathbb{Q}} \times \tau_{\mathbb{Q}} \mapsto \tau_{\mathbb{Q}}$, and predicate symbols $>, \geq$: $\mathbf{P}(\tau_{\mathbb{Q}} \times \tau_{\mathbb{Q}})$, as well as an arbitrary number of other function symbols. For every numeral multiplication $k \in \mathbb{Q} \setminus \{1\}$, we simply write k to denote the term $k(1)$ obtained by the numeral multiplication k applied to 1; in these cases, we refer to k as *numerals*. Throughout this paper, we use j, k, l to denote numerals, or numeral multiplications, possibly with indices.

We write $-t$ to denote the term $-1(t)$. If j, k are two numeral multiplications, by (jk) and $(j + k)$ we denote the numeral multiplication that corresponds to the result of multiplying and adding the rationals/numerals j and k, respectively. For applications of numeral multiplications $j(t)$ we may omit the parenthesis and write jt instead. If we write $+k$, or $-k$ for some numeral k, we assume k itself is positive. We write $\pm$ (and $\mp$) to denote either of the symbols $+$ or $-$ (and respectively $-$ or $+$). For $q \in \mathbb{Q}$ we define $\mathbf{sign}(q)$ to be 1 if $q > 0$, -1 if $q < 0$, and 0 otherwise. We call $+, \geq, >, 1$, and the numeral multiplications the $\mathbb{Q}$ *symbols*. Finally, an *atomic term* is either a logical variable, or the term 1, or a term whose top level function symbol is not a $\mathbb{Q}$ symbol.

A $\mathbb{Q}$-*model* interprets the sort $\tau_{\mathbb{Q}}$ as $\mathbb{Q}$, and all $\mathbb{Q}$ symbols as their corresponding functions/predicates on $\mathbb{Q}$. We write $\mathbb{Q} \models C$ iff for every $\mathbb{Q}$-model M, $M \models C$ holds. If $\mathcal{E}$ is a set of formulas, we call a model M a $\mathcal{E}$-*model* if $M \models \mathcal{E}$.

Term Orderings. We write $u[s]$ to denote that s is a subterm of u, where the subterm relation is denoted via $\trianglelefteq$. That is, $s \trianglelefteq u$; similar notation will also be used for literals $L[s]$ and clauses $C[s]$. We denote by $u[s \mapsto t]$ the term resulting from replacing all subterms s of u by t.

Multisets (of term, literals) are denoted with $\dot{\{} \dots \dot{\}}$. For a multiset S and natural number $n \in \mathbb{N}$, we define $0 * S = \emptyset$, and $n * S = (n - 1 * S) \cup S$ for $n > 0$.

Let $\prec$ be a relation and $\equiv$ be an equivalence relation. By $\prec^{\mathsf{mul}}_{\equiv}$ we denote the *multiset extension* of $\prec$, defined as the smallest relation satisfying $M \cup \dot{\{}s_1, \dots, s_n\dot{\}} \prec^{\mathsf{mul}}_{\equiv} N \cup \dot{\{}t\dot{\}}$, where $M \equiv N$, $n \geq 0$, and $s_i \prec t$ for $1 \leq i \leq n$. For $n, m \in \mathbb{N}$, by $\prec^{\mathsf{wmul}}_{\equiv}$ we denote the *weighted multiset extension*, defined by $\langle \frac{1}{n}, S\rangle \prec^{\mathsf{wmul}}_{\equiv} \langle \frac{1}{m}, T\rangle$ iff $m * S \prec^{\mathsf{mul}}_{\equiv} n * T$. We omit the equivalence relation $\equiv$ if it is clear in the context.

Let s, t, t_i be terms, θ, θ' be ground substitutions and $\mathcal{E}$ be a set of axioms. We write $s \equiv_{\mathcal{E}} t$ for $\mathcal{E} \models s \approx t$ and $\theta \equiv_{\mathcal{E}} \theta'$ iff for all variables x we have $x\theta \equiv_{\mathcal{E}} x\theta'$. We say that s is a $\mathcal{E}$-subterm of t ($s \trianglelefteq_{\mathcal{E}} t$) if $s \equiv_{\mathcal{E}} t$, or $t \equiv_{\mathcal{E}} f(t_1 \dots t_n)$ and $s \trianglelefteq_{\mathcal{E}} t_i$. We also say that s is a strict $\mathcal{E}$-subterm of t ($s \triangleleft_{\mathcal{E}} t$) if $s \trianglelefteq_{\mathcal{E}} t$ and $s \not\equiv_{\mathcal{E}} t$.

4 Theoretical Foundation for Unification with Abstraction

Our motivating example from Sect. 2 showcases that first-order arithmetic reasoning requires (i) establishing syntactic difference among terms (e.g. $2x$ and $x + 1$), while (ii) deriving they have instances that are semantically equal in models of a background theory $\mathcal{E}$ (e.g. the theory $\mathbb{Q}$).

A naive approach addressing (i)-(ii) would be to use an axiomatisation of the background theory $\mathcal{E}$, and use this axiomatisation for proof search in uninterpreted first-order logic. Such an approach can however be very costly. For example, even a relatively simple background theory **AC** axiomatizing commutativity and

```
1  fn uwa(s,t)
2      eqs ← {s ≈ t}; σ ← ∅; C ← ∅;
3      while eqs ≠ ∅
4          s ≈ t ← eqs.pop();
5          if s ≈ t ∈ {x ≈ u, u ≈ x} for some x ∈ V, x ⋪ u
6              ⟨σ, eqs, C⟩ ← ⟨σ ∪ {x ↦ u}, eqs, C⟩{x ↦ u};
7          else if canAbstract(s, t)
8              C.push(s ≉ t);
9          else if s = f(s_1 ... s_n), t = f(t_1 ... t_n)
10             eqs.push({s_1 ≈ t_1 ... s_n ≈ t_n})
11         else
12             return ⊥;
13     return ⟨σ, C⟩;
```

Algorithm 1: Computing an abstracting unifier uwa.

associativity of $\approx$, that is $\mathbf{AC} = \{x{+}y \approx y{+}x, x{+}(y{+}z) \approx (x{+}y){+}z\}$, would make a superposition-based theorem prover derive a vast amount of useless/redundant formulas as equational tautologies. An approach to circumvent such inefficient handling of equality reasoning is to use *unification modulo* **AC**, or in general *unification modulo* $\mathcal{E}$, as already advocated in [22, 34, 40]. In this section we describe the adjustments we made towards unification modulo $\mathcal{E}$, allowing us to introduce *unification with abstraction* (Sect. 4.1). We also show under which condition our method can be used to turn a complete superposition calculus using unification modulo $\mathcal{E}$ into a complete superposition calculus using unification with abstraction. Concretely, we show how this can be used for the specific theory of arithmetic $\mathcal{A}_{\mathsf{eq}}$ in the calculus Alasca (Sect. 4.2).

4.1 Unification with Abstraction – UWA

In a nutshell, unification modulo $\mathcal{E}$ finds substitutions σ that make two terms s, t equal in the background theory, i.e. $\mathcal{E} \models s\sigma \approx t\sigma$. While unification modulo $\mathcal{E}$ removes the need for axiomatisation of $\mathcal{E}$ during superposition reasoning, it comes with some inefficiencies. Most importantly, in contrast to syntactic unification, there is no unique most general unifier $\mathsf{mgu}(s, t)$ when unifying modulo $\mathcal{E}$ but only minimal complete sets of unifiers $\mathsf{mcu}_{\mathcal{E}}(s, t)$, which can be very large; for example, unification modulo **AC** is doubly exponential in general [22].

Bypassing the need for unification modulo $\mathcal{E}$, *fully abstracted clauses* are used in [40], without the need for axiomatisation of the theory $\mathcal{E}$ and without compromising completeness of the underlining superposition-based calculus. Our work extends ideas from [40] and adjusts *unification with abstraction* (uwa) from [34], allowing us to prove completeness of a calculus using uwa (Theorem 3).

Example 1. Let us first consider the example of factoring the clause $p(2x) \vee p(x{+}1)$, a simplified version of the unification step performed in line 5 in Fig. 1. That is, unifying the literals $p(2x)$ and $p(x + 1)$, in order to remove duplicate literals. Within the setting of [40], these literals would only exist in their fully abstracted

form, which can be obtained by replacing every subterm $t : \tau_{\mathbb{Q}}$ that is not a variable by a fresh variable x, and adding the constraint $x \not\approx t$ to the corresponding clause. Hence, the clause $p(2x) \vee p(x+1)$ is transformed to $p(y) \vee p(z) \vee y \not\approx 2x \vee z \not\approx x+1$ in [40]. Unification then becomes trivial: we would derive the clause $p(y) \vee y \not\approx 2x \vee y \not\approx x+1$ by factoring, from which $p(2x) \vee 2x \not\approx x+1$ is inferred using equality factoring and resolution.

Within unification with abstraction, we aim at cutting out intermediate steps of applying abstractions, equality resolution and factoring. As a result, we skip unnecessary consequences of intermediate clauses, and derive the conclusion $p(2x) \vee 2x \not\approx x+1$ straight away. To this end, we introduce constraints only for those $s, t : \tau_{\mathbb{Q}}$ on which unification fails. We thus gain the advantage that clauses are not present in the search space in their abstracted forms, increasing efficiency in proof search. Further, our unification with abstraction approach is parametrized by a predicate $\mathsf{canAbstract}$ to control the application of abstraction, as listed in Algorithm 1. This is yet another significant difference compared to fully abstracted clauses, as in the latter, abstraction is performed for every subterm $t : \tau_{\mathbb{Q}}$ without considering the terms with which t might be unified later.

Our uwa method can be seen as a lazy approach of full abstraction from [40]. We compute so-called abstracting unifiers $\mathsf{uwa}(s,t) = \langle \sigma, \mathcal{C} \rangle$ in Algorithm 1, allowing us to replace unification modulo $\mathcal{E}$ by unification with abstraction.

Definition 1 (Abstracting Unifier). *Let σ be a substitution and $\mathcal{C}$ a set of literals. A partial function* uwa *that maps two terms s, t either to $\bot$ or to a pair $\langle \sigma, \mathcal{C} \rangle = \mathsf{uwa}(s,t)$ is called an* abstracting unifier.

The abstracting unifier $\mathsf{uwa}(s,t)$ computed by Algorithm 1 is parametrized by the relation $\mathsf{canAbstract}$. The intuition of this relation is that $\mathsf{canAbstract}(s,t)$ holds for terms s and t, when $s \approx t$ might hold in the background theory $\mathcal{E}$. To ensure that unification with abstraction can replace unification modulo $\mathcal{E}$, we impose the following additional properties over the abstract unifier $\mathsf{uwa}(s,t)$.

Definition 2 (uwa Properties). *Let σ be a substitution and $\mathcal{C}$ a set of literals. Consider $s, t \in \mathbf{T}$ be such that $\mathsf{uwa}(s,t) = \langle \sigma, \mathcal{C} \rangle$ and let θ be an arbitrary ground substitution. We say* uwa *is*

- $\mathcal{E}$-sound *iff* $\mathcal{E} \models (s \approx t)\sigma \vee \mathcal{C}$;
- $\mathcal{E}$-general *iff* $\forall \mu \in \mathsf{mcu}_{\mathcal{E}}(s,t). \exists \rho. \sigma\rho \equiv_{\mathcal{E}} \mu$;
- $\mathcal{E}$-minimal *iff* $\mathcal{E} \models (s \approx t)\sigma\theta \implies \mathcal{E} \models (\neg\mathcal{C})\theta$;
- subterm-founded *with respect to the clause ordering $\prec$, iff for every uninterpreted function or predicate f, every literal $L[\circ]$, it holds that $\mathcal{E} \models (s \approx t)\theta \implies \mathcal{C}\theta \prec L[f(s)]\theta$ or $\mathcal{C}\theta \prec L[f(t)]\theta$.*

Further, uwa *is* $\mathcal{E}$-complete *if, for all $s, t \in \mathbf{T}$ with $\mathsf{uwa}(s,t) = \bot$, we have $\mathsf{mcu}_{\mathcal{E}}(s,t) = \emptyset$.*

Definition 2 is necessary to lift inferences using unification with abstraction. We thereby want to assure that, whenever C does not hold, then s and t are

equal; hence abstracting unifiers $\mathsf{uwa}(x, y) = \langle \emptyset, x + y \not\approx y + x \rangle$ would be unsound. The $\mathcal{E}$-generality property enforces that substitutions introduced by uwa are general enough in order to still be turned into a complete set of unifiers. As such, $\mathcal{E}$-generality is needed to rule out cases like $\mathsf{uwa}(x + y, 2) = \langle \{x \mapsto 0, y \mapsto 2\}, \emptyset \rangle$, which would not be able to capture, for example, the substitution $\{x \mapsto 1, y \mapsto 1\}$. We note that we use uwa to extend counterexample-reducing inference systems (see Definition 4), allowing inductive completeness proofs. As these inference systems need to derive conclusions that are smaller than the premises, we need the subterm-foundedness property to make sure to only introduce constraints that are smaller than the premises as well. If we have a look at the previous properties, we see that all of them are fulfilled if $\mathsf{uwa}(s, t) = \bot$. Therefore we need to make sure that uwa only returns $\bot$ when s and t are not unifiable modulo $\mathcal{E}$; this is captured by $\mathcal{E}$-completeness.

In addition to properties of abstract unifiers $\mathsf{uwa}(s, t)$, we also impose conditions over the canAbstract relation that parametrizes $\mathsf{uwa}(s, t)$. As Algorithm 1 only introduces equality constraints for subterm pairs that should be unified, a resulting abstracting unifier $\mathsf{uwa}(s, t)$ is sound. Further, under the assumption that the clause ordering is defined as in standard superposition (e.g. using multiset extensions of a simplification ordering that fulfills the subterm property), the abstracting unifier $\mathsf{uwa}(s, t)$ is also subterm-founded. However, to ensure that $\mathsf{uwa}(s, t)$ is also minimal, interpreted functions should not be treated as uninterpreted ones; hence the canAbstract relation needs to always trigger abstraction on interpreted functions. Finally, we require that canAbstract does not skip terms which are potentially equal modulo $\mathcal{E}$, in order to guarantee completeness. Hence, we define the following properties for canAbstract.

Definition 3 (canAbstract **Properties).** *Let* $s, t \in \mathbf{T}$. *The* canAbstract *relation*

- captures $\mathcal{E}$, *iff for all* s, t, *it holds that* $\exists \rho. \mathcal{E} \vDash (s \approx t)\rho \Longrightarrow \mathsf{canAbstract}(s, t)$;
- guards interpreted functions, *iff for all* s, t, *where* $\mathsf{sym}(s) = \mathsf{sym}(t)$ *is an interpreted function,* $\mathsf{canAbstract}(s, t)$ *holds.*

Based on the above, we derive the following result.

Theorem 1. *The abstracting unifier* uwa *computed by Algorithm 1 is subterm-founded and sound. If* canAbstract *guards interpreted functions, then* uwa *is* $\mathcal{E}$*-general and* $\mathcal{E}$*-minimal. If* canAbstract *guards interpreted functions and captures* $\mathcal{E}$, *then* uwa *is* $\mathcal{E}$*-complete.*

4.2 UWA Completeness

We now show how unification with abstraction (uwa) can be used to replace unification modulo $\mathcal{E}$ in saturation-based theorem proving [3]. We recall from [3] that in order to show refutational completeness of an inference-system Γ, one constructs a *model functor* I that maps sets of ground clauses N to candidate models I_N. In order to show that Γ is refutationally complete, one needs to show that if N is saturated with respect to Γ, then $I_N \vDash N$. For this, the notion of a counterexample-reducing inference system is introduced.

Definition 4. *We say an inference system Γ is* counterexample reducing*, with respect to a model functor I and a well-founded ordering on ground clauses $\prec$, if for every ground set of clauses N and every minimal $C \in N$ such that $I_N \not\models C$, there is an inference*

$$\frac{C_1 \quad \dots \quad C_n \quad C}{D}$$

where $\forall i. I_N \models C_i$, $\forall i. C_i \prec C$, $D \prec C$, and $I_N \not\models D$.

We then have the following key result.

Theorem 2 (Bachmair&Ganzinger [3]). *Let $\prec$ be a well-founded ordering on ground clauses and I be a model functor. Then, every inference system that is counterexample-reducing wrt $\prec$ and I is refutationally complete.*

This result also holds for an inference system being refutationally complete wrt $\mathcal{E}$ if for every N it holds that $I_N \models \mathcal{E}$. When constructing a refutationally complete calculus, one usually first defines a ground counterexample-reducing inference system and then lifts this calculus to a non-ground inference system. Lifting is done such that, if the ground inference system is counterexample reducing, then its lifted non-ground version is also counterexample reducing.

We next show how to transform a lifting of a counterexample-reducing inference system that uses unification modulo $\mathcal{E}$ into a lifting using unification with abstraction. That is, given a counterexample-reducing inference-system using unification modulo $\mathcal{E}$ to define its rules, we construct another counterexample-reducing inference system that uses uwa instead. As we only transform rules that use unification, we introduce the notion of a unifying rule.

Definition 5. *An inference rule γ is a* unifying rule *if it is of the form*

$$\frac{C_1 \quad \dots \quad C_n \quad C}{D\sigma}, \quad \text{where } \sigma \in \mathsf{mcu}_{\mathcal{E}}(s,t).$$

We also define the mapping $\circ_{\mathsf{uwa}}$ that maps unifying inferences γ to γ_{uwa} as

$$\gamma_{\mathsf{uwa}} = \left(\frac{C_1 \quad \dots \quad C_n \quad C}{D\sigma \vee \mathcal{C}}, \text{ where } \langle \sigma, \mathcal{C} \rangle = \mathsf{uwa}(s,t) \right)$$

Soundness of the unifying rule γ alone however does not suffice to show soundness of γ_{uwa}. Therefore we introduce a stronger notion of soundness that holds for all the rules we will consider to lift.

Definition 6. *Let γ be a unifying rule. We say γ is* strongly sound *iff $\mathcal{E}, C_1 \dots C_n, C \models s \approx t \to D$.*

Lemma 1. *Assume that γ is strongly sound and* uwa *is sound. Then, γ_{uwa} is sound.*

We note that not every inference can be transformed using $\circ_{\mathsf{uwa}}$, without compromising completeness. To circumvent this problem, we consider the notion of compatibility with respect to transformations.

Definition 7. *Let γ be a unifying inference. Then, γ* unifies strict subterms *iff for every grounding θ, $u \in \{s,t\}$ there is an uninterpreted function or predicate f, a literal $L[f(u)]$, and clause $C' \in \{C_1 \dots C_n, C\}$, such that $L[f(u)]\theta \preceq C'\theta$.*

Note that in the above definition we usually have that $L[f(s)]$ or $L[f(t)]$ is some literal of the premises.

Definition 8 (uwa-Compatibility). *We say an inference γ is* uwa compatible *if it is a unifying inference, strongly sound, and unifies strict subterms.*

Theorem 3. *Let* uwa *be a general, compatible, subterm-founded, complete, and minimal abstracting unifier. If Γ is the lifting of a counterexample-reducing inference system Γ^{ϑ} with respect to a model functor I, and clause ordering $\prec$, then $\Gamma_{\mathsf{uwa}} = \{\gamma_{\mathsf{uwa}} \mid \gamma \in \Gamma,\ \gamma$ is* uwa*-compatible$\} \cup \{\gamma \in \Gamma \mid\ \gamma$ is not* uwa*-compatible$\}$ is the lifting of an inference system $\Gamma^{\vartheta}_{\mathsf{uwa}}$ that is counterexample-reducing with respect to I and $\prec$.*

Theorem 1 and Theorem 3 together imply that, given a compatible inference system, we need to only specify the right canAbstract predicate in order to perform a lifting using uwa. In Sect. 5 we introduce the calculus Alasca, a concrete inference system with the desired properties, for which a suitable predicate canAbstract can easily be found.

5 ALASCA Reasoning

We use the lifting results of Sect. 4 to introduce our Alasca calculus for reasoning in quantified linear arithmetic, by combining superposition reasoning with Fourier-Motzkin type inference rules. While an instance of such a combination has been studied in the Lasca calculus of [26], Lasca is restricted to ground, i.e. quantifier-free, clauses. Our Alasca extends Lasca with uwa and provides an altered ground version Alasca$^{\theta}$ (Sect. 5.1) which efficiently can be lifted to the quantified domain (Sect. 5.2). As quantified reasoning with linear real arithmetic and uninterpreted functions is inherently incomplete, we provide formal guarantess about what Alasca can prove. Instead of focusing on completeness with respect to $\mathbb{Q}$-models as in [26], we show that Alasca is complete with respect to a partial axiomatisation $\mathcal{A}_{\mathbb{Q}}$ of $\mathbb{Q}$-models (Sect. 5.2).

5.1 The ALASCA Calculus – Ground Version

The Alasca calculus uses a partial axiomatisation $\mathcal{A}_{\mathbb{Q}}$ of $\mathbb{Q}$-models, and handles some $\mathbb{Q}$-axioms via inferences and some via uwa. We therefore split the axiom set $\mathcal{A}_{\mathbb{Q}}$ into $\mathcal{A}_{\mathsf{eq}}$ and $\mathcal{A}_{\mathsf{ineq}}$, as listed in Fig. 2.

Our Alasca calculus modifies the Lasca framework [26] to enable an efficient lifting for quantified reasoning. For simplicity, we first present the ground version of Alasca, which we refer to Alasca$^{\theta}$, whose one key benefit is illustrated next.

$$\mathcal{A}_{\mathbb{Q}} = \mathcal{A}_{\mathsf{eq}} \cup \mathcal{A}_{\mathsf{ineq}}$$

$$\mathcal{A}_{\mathsf{eq}} = \mathbf{AC} \cup \{jx + kx \approx (j+k)x \mid j,k \in \mathbb{Q}\} \cup \{j(k(x)) \approx (jk)x \mid j,k \in \mathbb{Q}\} \cup \{1(x) \approx x\} \cup \{k(x+y) \approx kx + ky \mid k \in \mathbb{Q}\} \cup \{x + 0 \approx x, 0x \approx 0\}$$

$$\mathcal{A}_{\mathsf{ineq}} = \{x > y \wedge y > z \to x > z\} \cup \{x > y \to x + z > y + z\} \cup \{x > y \vee x \approx y \vee y > x\} \cup \{\neg(x > x)\} \cup \{x \geq y \leftrightarrow (x > y \vee x \approx y)\} \cup \{x > y \to +kx > +ky \mid +k \in \mathbb{Q}\} \cup \{x > y \to -ky > -kx \mid -k \in \mathbb{Q}\}$$

Fig. 2. Axioms handled by the Alasca calculus. All are implicity universally quantified.

Example 2. One central rule of Alasca is the Fourier-Motzkin variable elimination rule (FM). We use (FM) in line 7 of Fig. 1, when proving the motivating example of Sect. 2, given in formula (1). Namely, using (FM), we derive $-2x-y+sk > 0$ from $f(2x,y) - 2x - y > 0$ and $-f(2,y) + sk \geq 0$, under the assumption that $2x \approx 2$. The (FM) rule can be seen as a version of the inequality chaining rules of [3] , chaining the inequalities $sk \geq f(2,y)$ and $f(2x,y) > 2x + y$. Moreover, the (FM) rule can also be considered a version of binary resolution, as it resolves the positive summand $f(2x,y)$ with the negative summand $-f(2,y)$, mimicing thus resolution over subterms, instead of literals. The main benefit of (FM) comes with its restricted application to maximal atomic terms in a sum (instead of its naive application whenever possible).

Alasca$^\theta$ Normalization and Orderings. Compared to Lasca [26], the major difference of Alasca$^\theta$ comes with focusing on which terms are being considered equal within inferences; this in turn requires careful adjustments in the underlying orderings and normalization steps of Alasca$^\theta$, and later also in unification within Alasca. In Lasca terms are rewritten in their so-called $\mathbb{Q}$-normalized form, while equality inference rules exploit equivalence modulo **AC**. Lifting such inference rules is however tricky. Consider for example the application of the rewrite rule $j(ks) \to (jk)s$ (triggered by $j(ks) \approx (jk)s$) over the clause $C[jx, x]$. In order to lift all instances of this rewrite rule, we would need to derive $C[(jk)x, kx]$ for every $k \in \mathbb{Q}$, which would yield an infinite number of conclusions. In order to resolve this matter, Alasca$^\theta$ takes a different approach to term normalization and handling equivalence. That is, unlike Lasca, we formulate all inference rules using equivalence modulo $\mathcal{A}_{\mathsf{eq}}$, and do not consider the normalization of terms as simplification rules.

As Alasca$^\theta$ rules use equivalence modulo $\mathcal{A}_{\mathsf{eq}}$, we also need to impose that the simplification ordering used by Alasca$^\theta$ is $\mathcal{A}_{\mathsf{eq}}$-compatible. Intuitively, $\mathcal{A}_{\mathsf{eq}}$-compatibility means that terms that are equivalent modulo $\mathcal{A}_{\mathsf{eq}}$ are in one equivalence class wrt the ordering. This allows us to replace terms by an arbitrary normal form wrt these equivalence classes before and after applying any inference rules, allowing it to use a normalization similar to $\mathbb{Q}$-normalization that does not need to be lifted. Hence, we introduce $\mathcal{A}_{\mathsf{eq}}$*-normalized terms* as being terms

whose sort is not $\tau_{\mathbb{Q}}$ or of the form $\frac{1}{k}(k_1 t_1 + \cdots + k_n t_n)$, such that $\forall i. k_i \in \mathbb{Z} \setminus 0$, $\forall i \neq j. t_i \not\equiv t_j$, $\forall i. t_i$ is atomic, k is positive, and $\gcd(\{k, k_1 \ldots k_n\}) = 1$. Obviously every term can be turned into a $\mathcal{A}_{\mathsf{eq}}$-normalized term. For the rest of this section we assume terms are $\mathcal{A}_{\mathsf{eq}}$-normalized, and write $\equiv$ for $\equiv_{\mathcal{A}_{\mathsf{eq}}}$. We also assume that literals with interpreted predicates $\diamond$ are being normalized (during preprocessing) and to be of the form $t \diamond 0$. We write $s \mathrel{\hat{\approx}} t$ for equalities, with sorts different from $\tau_{\mathbb{Q}}$, and for equalities of sort $\tau_{\mathbb{Q}}$ that can be rewritten to $s \approx t$ such that s is an atomic term. Finally, Alasca$^{\theta}$ also extends Lasca by not only handling the predicates $>$ and $\approx$, but also $\geq$, and $\not\approx$, which has the advantage that inequalities are not being introduced in purely equational problems in Alasca$^{\theta}$.

As discussed in Example 2, the (FM) rule of Alasca$^{\theta}$ is similar to binary resolution, as it can be seen as "resolving" atomic subterms instead of literals. To formalize such handling of terms in (FM), we distinguish so-called $\mathsf{atoms}(t)$, atoms of some term t. Doing so, given an $\mathcal{A}_{\mathsf{eq}}$-normalized term $t = \frac{1}{k}(\pm_1 k_1 t_1 + \ldots \pm_n k_n t_n)$, we define $\mathsf{atoms}^{\pm}(t) = \langle k, k_1 * \dot{\{} \pm_1 t_1 \dot{\}} \cup \ldots \cup k_n * \dot{\{} \pm_n t_n \dot{\}} \rangle$ and $\mathsf{atoms}(t) = \langle k, k_1 * \dot{\{} t_1 \dot{\}} \cup \ldots \cup k_n * \dot{\{} t_n \dot{\}} \rangle$. We extend both of these functions $f \in \{\mathsf{atoms}, \mathsf{atoms}^{\pm}\}$ to literals as follows: $f(t \diamond 0) = f(t)$, assuming that the term t has been normalised to $\frac{1}{k} = 1$ before. For (dis)equalities $s \approx t$ ($s \not\approx t$) of uninterpreted sorts, we define atoms to be $\langle 1, \dot{\{} s, t \dot{\}} \rangle$. Further we define $\mathsf{maxAtoms}(t)$, to be the set of maximal terms in $\mathsf{atoms}(t)$ with respect $\prec$, and $\mathsf{maxAtom}(t) = t_0$ if $\mathsf{maxAtoms}(t) = \{t_0\}$.

Alasca$^{\theta}$ Inferences. The inference rules of Alasca$^{\theta}$ are summarized in Fig. 3a. All rules are parametrized by a $\mathcal{A}_{\mathsf{eq}}$-compatible ordering relation $\prec$ on ground terms, literals and clauses. Underlining a literal in a clause or an atomic term in a sum means that the underlined expression is non-strictly maximal wrt to the other literals in the clause, or atomic terms in the sum. We use double-underlining to denote that the expression is strictly maximal. We call $\mathbf{L}^{\theta}_{+}$ the set of potentially productive literals, defined as all equalities and inequalities with strictly maximal atomic term with positive coefficient.

Finding a right ordering relation is non-trivial, as many different requirements, like compatibility, subterm property, well-foundedness, and stability under substitutions, need to be met [25, 26, 39, 41]. For Alasca, we use a modified version of the Qkbo ordering of [26], with the following two modifications.

(i) Firstly, the Alasca ordering is defined for non-ground terms. This means that the ordering needs to handle subterms with sums where there is no maximal atomic summand, like the term $x + y$. In addition, our ordering needs to be stable under substitutions in order to work with non-ground terms. Note however that our atom functions atoms and $\mathsf{atoms}^{\pm}$ are not stable under substitutions, as the term $f(x) - f(y)$ and the substitution $\{x \mapsto y\}$ demonstrates. Therefore, we parametrize our Alasca ordering by the relation $\mathsf{subsSafe}$. The $\mathsf{subsSafe}$ relation fulfils the property that if $\mathsf{subsSafe}(\frac{1}{k}(\pm_1 k_1 t_1 + \cdots \pm_n k_n t_n))$, then there is no substitution θ such that $\pm_i k_i t_i \theta \equiv \mp_j k_j t_j \theta$, for any i, j. In general, checking the existence of such a θ is as hard as unifying modulo $\mathcal{A}_{\mathsf{eq}}$. Nevertheless, we can overapproximate the $\mathsf{subsSafe}$ relation using the $\mathsf{canAbstract}$ predicate.

Fourier-Motzkin Elimination

$$\frac{C_1 \vee +j\underline{s} + t_1 \gtrsim_1 0 \qquad C_2 \vee -k\underline{s'} + t_2 \gtrsim_2 0}{C_1 \vee C_2 \vee kt_1 + jt_2 > 0} \ (\mathsf{FM})$$

where
- $js + t_1 > 0 \succ C_1$
- $-ks' + t_2 > 0 \succeq C_2$
- $s \equiv s'$
- $\{>\} \subseteq \{\gtrsim_1, \gtrsim_2\} \subseteq \{>, \geq\}$

Tight Fourier-Motzkin Elimination

$$\frac{C_1 \vee +j\underline{s} + t_1 \geq 0 \qquad C_2 \vee -k\underline{s'} + t_2 \geq 0}{C_1 \vee C_2 \vee kt_1 + jt_2 > 0 \vee -ks' + t_2 \approx 0} \ (\mathsf{FM}^{\geq})$$

where
- $js + t_1 > 0 \succ C_1$
- $-ks' + t_2 > 0 \succeq C_2$
- $s \equiv s'$

Inequality Factoring

$$\frac{C \vee +j\underline{s} + t_1 \gtrsim_1 0 \vee +k\underline{s'} + t_2 \gtrsim_2 0}{C \vee kt_1 - jt_2 \gtrsim_3 0 \vee +ks' + t_2 \gtrsim_2 0} \ (\mathsf{IF})$$

where
- $s \equiv s'$
- $\forall L \in (C \vee js \ + t_1 \gtrsim_1 0).ks' + t_2 \gtrsim_2 0 \succeq L$ or $\forall L \in (C \vee ks' + t_2 \gtrsim_2 0).js \ + t_1 \gtrsim_1 0 \succeq L$
- $\gtrsim_i \in \{>, \geq\}$
- $\gtrsim_3 = \begin{cases} \geq & \text{if } \gtrsim_1 = \geq, \text{ and } \gtrsim_2 = > \\ > & \text{else} \end{cases}$

Term Factoring

$$\frac{C \vee j\underline{s} + k\underline{s'} + t \diamond 0}{C \vee (j+k)s' + t \diamond 0} \ (\mathsf{TF})$$

where
- $s \equiv s'$
- $\diamond \in \{>, \geq, \approx, \not\approx\}$
- $s, s' \in \mathsf{maxAtoms}(C \vee js + ks' + t \diamond 0)$
- there is no uninterperted literal in C

Contradiction

$$\frac{C \vee \cancel{\pm k \diamond 0}}{C} \ (\mathsf{Triv})$$

where
- $\diamond \in \{>, \geq, \approx, \not\approx\}$
- $k \in \mathbb{Q}$
- $\mathbb{Q} \not\models \pm k \diamond 0$

Superposition

$$\frac{C_1 \vee \underline{s} \approx t \qquad C_2 \vee L[s']}{C_1 \vee C_2 \vee L[s' \to t]} \ (\ \mathsf{up})$$

where
- $s \equiv s'$
- $s \hat{\approx} t \succ C_1$
- $L[s'] \in \mathbf{L}^{\theta}_{+}$ & $L[s'] \succ C_2$ or $L[s'] \notin \mathbf{L}^{\theta}_{+}$ & $L[s'] \succeq C_2$
- $s' \trianglelefteq x \in \mathsf{maxAtoms}(L[s'])$
- $s \approx t \vee C_1 \prec C_2 \vee L[s']$

Equality Resolution

$$\frac{C \vee s \not\approx s'}{C} \ (\mathsf{ER})$$

where
- $s \equiv s'$
- $s \not\approx s' \succeq C$

Equality Factoring

$$\frac{C \vee \underline{s} \hat{\approx} t_1 \vee \underline{s'} \hat{\approx} t_2}{C \vee t_1 \not\approx t_2 \vee s \approx t_1} \ (\mathsf{EF})$$

where
- $s \equiv s'$
- $s' \approx t_2 \succeq C \vee s \approx t_1$

(a) Rules of the ground calculus Alasca$^{\theta}$.

Variable Elimination

$$\frac{C \vee \bigvee_{i \in I} x + b_i \gtrsim_i 0 \vee \bigvee_{j \in J} -x + b_j \gtrsim_j 0 \vee \bigvee_{k \in K} x + b_k \approx 0 \vee \bigvee_{l \in L} x + b_l \not\approx 0}{\bigwedge_{K^+ \subseteq K} \left(\begin{array}{l} C \vee \bigvee_{i \in I, j \in J} b_i + b_j \gtrsim_{i,j} 0 \vee \bigvee_{i \in I, k \in K^-} b_i - b_k \geq 0 \vee \bigvee_{i \in I, l \in L} b_i - b_l \gtrsim_i 0 \\ \qquad \vee \bigvee_{j \in J, k \in K^+} b_j + b_k \geq 0 \vee \bigvee_{j \in J, l \in L} b_j + b_l \gtrsim_j 0 \\ \qquad \vee \bigvee_{k_1 \in K^+, k_2 \in K^-} b_{k_1} - b_{k_2} \geq 0 \vee \bigvee_{k \in K^+, l \in L} b_k - b_l \geq 0 \\ \qquad \vee \bigvee_{k \in K^-, l \in L} b_l - b_k \geq 0 \\ \qquad \vee \bigvee_{l_1, l_2 \in L} b_{l_1} - b_{l_2} \not\approx 0 \end{array} \right)} \ (\mathsf{VE})$$

where
- x is an unshielded variable
- $K^- = K \setminus K^+$
- C does not contain x
- $\gtrsim_i, \gtrsim_j \in \{\geq, >\}$
- $(\gtrsim_{i,j}) = \begin{cases} (\geq) & \text{if } \geq \in \{\gtrsim_i, \gtrsim_j\} \\ (>) & \text{otherwise} \end{cases}$

(b) Variable elimination rule used for lifting Alasca$^{\theta}$.

Fig. 3. Inference rules used to define the calculus Alasca.

(ii) Secondly, we adjusted the ALASCA ordering to be $\mathcal{A}_{\mathsf{eq}}$-compatible, instead of **AC**-compatible. We modified the literal ordering of ALASCA, such that literals are ordered by all their atoms using the weighted multiset extension of $\prec$, instead of only using the maximal one of each literal L as in [26].

We define a model functor $\mathcal{I}_\infty$ mapping clauses to $\mathcal{A}_{\mathbb{Q}}$-models (see [23] for details) and conclude the following.

Theorem 4. *ALASCA$^\theta$ is a counterexample-reducing inference system with respect to $\mathcal{I}_\infty$ and $\prec$.*

5.2 ALASCA Lifting and Completeness

Variable Elimination. Theorem 4 establishes completeness of ALASCA$^\theta$ for ground clauses wrt $\mathcal{A}_{\mathbb{Q}}$. We next lift this result (and calculus) to non-ground clauses.

We introduce the concept of an *unshielded variable*. We say a term $t : \tau_{\mathbb{Q}}$ is a top level term of a literal L if $t \in \mathsf{atoms}(L)$. We call a variable x *unshielded* in some clause C if x is a top level term of a literal in C, and there is no literal with an atomic top level term $t[x]$. Observe that within the ALASCA$^\theta$ rules, only maximal atomic terms in sums are being used in rule applications. This means, lifting ALASCA$^\theta$ to ALASCA is straightforward for clauses where all maximal terms in sums are not variables. Further, due to the subterm property, if a variable is maximal in a sum then it must be unshielded. Hence, the only variables we have to deal within ALASCA rule applications are unshielded ones.

The work of [40] modifies a standard saturation algorithm by integrating it with a variable elimination rule that gets rid of unshielded variables, without compromising completeness of the calculus. Based on [40] and the variable elimination rule of [3], we extend ALASCA$^\theta$ with the Variable Elimination Rule (VE), as given in Fig. 3b. In what follows, we show that the handling of unshielded variables in Fig. 3b can naturally be done within a standard saturation framework.

The (VE) rules replaces any clause with a set of clauses that is equivalent and does not contain unshielded variables. We assume that the clause is normalized, such that in every inequality x only occurs once with a factor 1 or -1, whereas for for equalities, x only occurs with factor 1. A simple example for the application of (VE) is the clause $a - x > 0 \vee x - b > 0 \vee a + b + x \geq 0$, where $x \in \mathbf{V}$, and a, b are constants. By reasoning about inequalities, it is easy to see that this is equivalent to $a > x \vee a + b \geq x \vee x > b$, thus further equivalent to $a > b \vee a + b \geq b$, which illustrates the benefit of variable elimination through (VE).

Lemma 2. *The conclusion of (VE) is equivalent to its premise.*

ALASCA Calculus - Non-Ground Version with Unification with Abstraction. We now define our lifted calculus ALASCA, as follows. Let ALASCA$^-$ be the calculus ALASCA$^\theta$ being lifted for clauses without unshielded variables. We define ALASCA to be ALASCA$^-$ chained with the variable elimination rule. That is, the result of every rule application is simplified using (VE) as long as applicable.

Theorem 5. *Alasca is the lifting of a counterexample-reducing inference system for sets of clauses without unshielded variables.*

Theorem 5 implies that Alasca is refutationally complete wrt $\mathcal{A}_{\mathbb{Q}}$ for sets of clauses without unshielded variables. As (VE) can be used to preprocess arbitrary sets of clauses to eliminate all unshielded variables, we get the following.

Corollary 1. *If N is a set of clauses that is unsatisfiable with respect to $\mathcal{A}_{\mathbb{Q}}$, then N can be refuted using Alasca.*

We conclude this section by specifying the lifting of Alasca$^{\theta}$ to get Alasca^{-}. To this end, we use our uwa results and properties for unification with abstraction (Sect. 4). We note that using unification modulo $\mathcal{A}_{\mathsf{eq}}$ would require us to develop an algorithmic approach that computes a complete set of unifiers modulo $\mathcal{A}_{\mathsf{eq}}$, which is a quite challenging task both in theory and in practice. Instead, using Theorem 1 and Theorem 3, we need to only specify a canAbstract predicate that guards interpreted functions and captures $\mathcal{A}_{\mathsf{eq}}$ within uwa. This is achieved by defining $\mathsf{canAbstract}(s,t)$ if any function symbol $f \in \{\mathsf{sym}(s), \mathsf{sym}(t)\}$ is an interpreted function $f \in \mathbb{Q} \cup \{+\}$.This choice of the canAbstract predicate is a slight modification of the abstraction strategy `one_side_interpreted` of [34]. We note that this is not the only choice for the predicate to fulfil the canAbstract properties. Consider for example the terms $f(x) + a$, and $a + b$. There is no substitution that will make these two terms equal, but our abstraction predicate introduces a constraint upon trying to unify them. In order to address this, we introduce an alternative canAbstract predicate that compares the atoms of a term, instead of only looking at the outer most symbol (Sect. 6).

We believe more precise abstraction predicates can improve proof search, as evidenced by our experiments using second abstraction predicate (Sect. 6).

6 Implementation and Experiments

We implemented Alasca [5] in the extension of the Vampire theorem prover [27].

Benchmarks. We evaluated the practicality of Alasca using the following six sets of benchmarks, resulting all together in 6374 examples, as listed in Table 1 and detailed next. (i) We considered all sets of benchmarks from the SMT-LIB repository [7] set that involve real arithmetic and uninterpreted functions, but no other theories. These are the three benchmark sets corresponding to the LRA, NRA, and UFLRA logics in SMT-LIB. (ii) We further used Sledgehammer examples generated by [15], using the SMT-LIB syntax. From the examples of [15], we selected those benchmarks that involve real arithmetic but no other theories. We refer to this benchmark set as SH. (iii) Finally, we also created two new sets of benchmarks, Triangular, and Limit, exploiting various mathematical properties. The Triangular suite contains variations of our motivating example from Sect. 2, and thus comes with reasoning challenges about triangular inequalities

[5] available at https://github.com/vprover/vampire/tree/alasca

Benchmarks (#)	Alasca	Cvc5	Vampire	Yices	UltElim	SmtInt	veriT	solved
all (6374)	**5744**	5626	5585	5531	5218	828	465	5988
LRA (1722)	1572	1401	1396	**1722**	1469	623	89	1722
NRA (3814)	3800	3804	3803	**3809**	3669	0	0	3812
UFLRA (10)	**10**	**10**	**10**	0	0	**10**	**10**	10
Triangular (34)	**24**	10	13	0	0	0	6	25
Limit (280)	**100**	90	81	0	80	0	90	100
SH (514)	238	**311**	282	0	0	195	270	319

Table 1. Experimental results, showing the numbers of solved problems.

and continuous functions. The Limit benchmark set is comprised of problems that combine various limit properties of real-valued functions.

Experimental Setup. We compared our implementation against the solvers from the `Arith` (arithmetic) division of the SMT-COMP competition 2022. These solvers, given in columns 3–8 of Table 1, are: Cvc5 [5], Vampire [35], Yices [19], UltElim [8], SmtInt [21], and veriT [2]. We note that Vampire is run in its competition portfolio mode, which includes the work from [34]. Alasca uses the same portfolio but implements our modified version of unification with abstraction (Sect. 4), disabling the use of theory axioms relying on our new Alasca rules (Sect. 5). We ran our experiments using the SMT-COMP 2022 competition setup: based on the StarExec Iowa cluster, with a 20 minutes timeout and using 4 cores. Benchmarks, solvers and results are publicly available[6].

Experimental Results. Table 1 summarizes our experimental findings and indicates the overall best performance of Alasca. For example, Alasca outperforms the two best arithmetic solvers of SMT-COMP 2022 by solving 118 more problems than Cvc5 and 159 more problems than Vampire.

7 Conclusions and Future Work

We introduced the Alasca calculus and drastically improved the performance of superposition theorem proving on linear arithmetic. Alasca eliminates the use of theory axioms by introducing theory-specific rules such as an analogue of Fourier-Motzkin elimination. We perform unification with abstraction with a general theoretical foundation, which, together with our variable elimination rules, serves as a replacement for unification modulo theory. Our experiments show that Alasca is competitive with state-of-the-art theorem provers, solving more problems than any prover that entered the arithmetic division in SMT-COMP 2022. Future work includes designing an integer version of Alasca, developing different versions for the canAbstract predicate, and improving literal/clause selections within Alasca.

Acknowledgements. This work was partially supported by the ERC Consolidator Grant ARTIST 101002685, the TU Wien Doctoral College SecInt, the FWF SFB project SpyCoDe F8504, and the EPSRC grant EP/V000497/1.

[6] https://www.starexec.org/starexec/secure/explore/spaces.jsp?id=535817

References

1. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity Compiler's Model Checker. In: CAV, LNCS, vol. 13371, pp. 325–338, Springer (2022), https://doi.org/10.1007/978-3-031-13185-1_16
2. Andreotti, B., Barbosa, H., Fontaine, P., Schurr, H.J.: veriT at SMT-COMP 2022. https://smt-comp.github.io/2022/system-descriptions/veriT.pdf (2022)
3. Bachmair, L., Ganzinger, H.: Ordered Chaining Calculi for First-Order Theories of Transitive Relations. J. ACM **45**(6), 1007–1049 (1998), https://doi.org/10.1145/293347.293352, URL https://doi.org/10.1145/293347.293352
4. Bachmair, L., Ganzinger, H.: Resolution Theorem Proving. In: Handbook of Automated Reasoning, pp. 19–99, Elsevier and MIT Press (2001), https://doi.org/10.1016/b978-044450813-3/50004-7
5. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., , Zohar, Y.: CVC5 at the SMT Competition 2022. https://smt-comp.github.io/2022/system-descriptions/cvc5.pdf (2022)
6. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A Versatile and Industrial-Strength SMT Solver. In: TACAS, LNCS, vol. 13243, pp. 415–442, Springer (2022), https://doi.org/10.1007/978-3-030-99524-9_24
7. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
8. Barth, M., Dietsch, D., Heizmann, M., Podelski, A.: Ultimate Eliminator at SMT-COMP 2022. https://smt-comp.github.io/2022/system-descriptions/UltimateEliminator%2BMathSAT.pdf (2022)
9. Baumgartner, P., Bax, J., Waldmann, U.: Beagle - A Hierarchic Superposition Theorem Prover. In: CADE, LNCS, vol. 9195, pp. 367–377, Springer (2015), https://doi.org/10.1007/978-3-319-21401-6_25
10. Bonacina, M.P., Graham-Lengrand, S., Shankar, N.: Satisfiability Modulo Theories and Assignments. In: CADE, LNCS, vol. 10395, pp. 42–59, Springer (2017), https://doi.org/10.1007/978-3-319-63046-5_4
11. Bromberger, M., Fleury, M., Schwarz, S., Weidenbach, C.: SPASS-SATT - A CDCL(LA) solver. In: CADE, LNCS, vol. 11716, pp. 111–122, Springer (2019), https://doi.org/10.1007/978-3-030-29436-6_7
12. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: TACAS, LNCS, vol. 6015, pp. 150–153, Springer (2010), https://doi.org/10.1007/978-3-642-12002-2_12
13. Cook, B.: Formal Reasoning About the Security of Amazon Web Services. In: CAV, LNCS, vol. 10981, pp. 38–47, Springer (2018), https://doi.org/10.1007/978-3-319-96145-3_3
14. Cruanes, S.: Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. Ph.D. thesis, Ecole Polytechnique, Paris, France (2015)
15. Desharnais, M., Vukmirovic, P., Blanchette, J., Wenzel, M.: Seventeen Provers Under the Hammer. In: ITP, LIPIcs, vol. 237, pp. 8:1–8:18 (2022), https://doi.org/10.4230/LIPIcs.ITP.2022.8
16. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling Static Analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019), https://doi.org/10.1145/3338112

17. Duarte, A., Korovin, K.: Implementing Superposition in iProver (System Description). In: IJCAR, LNCS, vol. 12167, pp. 388–397, Springer (2020), https://doi.org/10.1007/978-3-030-51054-1_24
18. Elad, N., Rain, S., Immerman, N., Kovács, L., Sagiv, M.: Summing up Smart Transitions. In: CAV, LNCS, vol. 12759, pp. 317–340, Springer (2021), https://doi.org/10.1007/978-3-030-81685-8_15
19. Graham-Lengrand, S.: Yices-QS 2022, an extension of Yices for quantified satisfiability. https://smt-comp.github.io/2022/system-descriptions/YicesQS.pdf (2022)
20. Gurfinkel, A.: Program Verification with Constrained Horn Clauses (Invited Paper). In: CAV, LNCS, vol. 13371, pp. 19–29, Springer (2022), https://doi.org/10.1007/978-3-031-13185-1_2
21. Hoenicke, J., Schindler, T.: SMTInterpol with Resolution Proofs. https://smt-comp.github.io/2022/system-descriptions/smtinterpol.pdf (2022)
22. Kapur, D., Narendran, P.: Double-exponential Complexity of Computing a Complete Set of AC-Unifiers. In: LICS, pp. 11–21, IEEE Computer Society (1992), https://doi.org/10.1109/LICS.1992.185515
23. Korovin, K., Kovács, L., Schoisswohl, J., Reger, G., Voronkov, A.: ALASCA:Reasoning in Quantified Linear Arithmetic (Extended Version). EasyChair Preprint no. 9606 (2023)
24. Korovin, K., Tsiskaridze, N., Voronkov, A.: Conflict Resolution. In: CP, LNCS, vol. 5732, pp. 509–523, Springer (2009), https://doi.org/10.1007/978-3-642-04244-7_41
25. Korovin, K., Voronkov, A.: An AC-Compatible Knuth-Bendix Order. In: CADE, LNCS, vol. 2741, pp. 47–59, Springer (2003), https://doi.org/10.1007/978-3-540-45085-6_5
26. Korovin, K., Voronkov, A.: Integrating Linear Arithmetic into Superposition Calculus. In: CSLs, LNCS, vol. 4646, pp. 223–237, Springer (2007), https://doi.org/10.1007/978-3-540-74915-8_19
27. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: CAV, LNCS, vol. 8044, pp. 1–35, Springer (2013), https://doi.org/10.1007/978-3-642-39799-8_1
28. de Moura, L.M., Bjørner, N.S.: Efficient E-Matching for SMT Solvers. In: CADE, LNCS, vol. 4603, pp. 183–198, Springer (2007), https://doi.org/10.1007/978-3-540-73595-3_13
29. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS, LNCS, vol. 4963, pp. 337–340, Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
30. de Moura, L.M., Jovanovic, D.: A Model-Constructing Satisfiability Calculus. In: VMCAI, LNCS, vol. 7737, pp. 1–12, Springer (2013), https://doi.org/10.1007/978-3-642-35873-9_1
31. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Handbook of Automated Reasoning, pp. 371–443, Elsevier and MIT Press (2001), https://doi.org/10.1016/b978-044450813-3/50009-6
32. Passmore, G.O.: Some Lessons Learned in the Industrialization of Formal Methods for Financial Algorithms. In: FM, LNCS, vol. 13047, pp. 717–721, Springer (2021), https://doi.org/10.1007/978-3-030-90870-6_39
33. Reger, G., Bjørner, N.S., Suda, M., Voronkov, A.: AVATAR Modulo Theories. In: GCAI, EPiC Series in Computing, vol. 41, pp. 39–52, EasyChair (2016), https://doi.org/10.29007/k6tp
34. Reger, G., Suda, M., Voronkov, A.: Unification with Abstraction and Theory Instantiation in Saturation-Based Reasoning. In: TACAS, LNCS, vol. 10805, pp. 3–22, Springer (2018), https://doi.org/10.1007/978-3-319-89960-2_1
35. Reger, G., Suda, M., Voronkov, A., Kovács, L., Bhayat, A., Gleiss, B., Hajdu, M., Hozzova, P., Evgeny Kotelnikov, J.R., Rawson, M., Riener, M., Robillard, S.,

Schoisswohl, J.: Vampire 4.7-SMT System Description. https://smt-comp.github.io/2022/system-descriptions/Vampire.pdf (2022)
36. Reynolds, A., King, T., Kuncak, V.: Solving Quantified Linear Arithmetic by Counterexample-Guided Instantiation. FMSD **51**(3), 500–532 (2017), https://doi.org/10.1007/s10703-017-0290-y
37. Schulz, S., Cruanes, S., Vukmirovic, P.: Faster, Higher, Stronger: E 2.3. In: CADE, LNCS, vol. 11716, pp. 495–507, Springer (2019), https://doi.org/10.1007/978-3-030-29436-6_29
38. Voronkov, A.: AVATAR: The Architecture for First-Order Theorem Provers. In: CAV, LNCS, vol. 8559, pp. 696–710, Springer (2014), https://doi.org/10.1007/978-3-319-08867-9_46
39. Waldmann, U.: Extending Reduction Orderings to ACU-Compatible Reduction Orderings. Inf. Process. Lett. **67**(1), 43–49 (1998), https://doi.org/10.1016/S0020-0190(98)00084-2
40. Waldmann, U.: Superposition for Divisible Torsion-Free Abelian Groups. In: CADE, LNCS, vol. 1421, pp. 144–159, Springer (1998), https://doi.org/10.1007/BFb0054257
41. Yamada, A., Winkler, S., Hirokawa, N., Middeldorp, A.: AC-KBO Revisited. Theory Pract. Log. Program. **16**(2), 163–188 (2016), https://doi.org/10.1017/S1471068415000083

A Matrix-Based Approach to Parity Games

Saksham Aggarwal, Alejandro Stuckey de la Banda, Luke Yang, and Julian Gutierrez(✉)

Monash University, Faculty of Information Technology, Melbourne, Australia
{sagg0005,astu0006,lyan0042}@student.monash.edu
julian.gutierrez@monash.edu

Abstract. Parity games are two-player zero-sum games of infinite duration played on finite graphs for which no solution in polynomial time is still known. Solving a parity game is an NP ∩ co-NP problem, with the best worst-case complexity algorithms available in the literature running in quasi-polynomial time. Given the importance of parity games within automated formal verification, several practical solutions have been explored showing that considerably large parity games can be solved somewhat efficiently. Here, we propose a new approach to solving parity games guided by the efficient manipulation of a suitable matrix-based representation of the games. Our results show that a sequential implementation of our approach offers very competitive performance, while a parallel implementation using GPUs outperforms the current state-of-the-art techniques. Our study considers both real-world benchmarks of structured games as well as parity games randomly generated. We also show that our matrix-based approach retains the optimal complexity bounds of the best recursive algorithm to solve large parity games in practice.

Keywords: Parity games · Formal verification · Parallel computing.

1 Introduction

Parity games are one of the most useful and effective algorithmic tools used in automated formal verification [18,5,2]. Indeed, several computational problems, such as model checking and automated synthesis using temporal logic specifications, can be reduced to the solution of a parity game [5,2]. More formally, a parity game is a two-player zero-sum game of infinite duration played on a finite graph. Since these games are determined [14,8], solving them is equivalent to finding a winning strategy for one of the two players in the game; or, similarly, deciding from which vertices in the graph one of the two players in the game can force a win no matter the strategy that the other player makes use of. The main question regarding parity games is that of the computational complexity of finding a solution of the game, a problem that is known to be in NP ∩ co-NP [11]. However, despite decades of research, a polynomial-time algorithm to solve such games remains elusive. The best-known decision procedures to solve parity games, most of them recently developed [4,13], run in quasi-polynomial time, which provide better worst-case complexity upper bounds than previous exponential-time approaches [18] found in the parity games literature.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 666–683, 2023.
https://doi.org/10.1007/978-3-031-30823-9_34

The importance of parity games in the solution of real-life automated verification problems, and the lack of a polynomial-time decision procedure to solve such games, has motivated the development and implementation of algorithms that can solve parity games somewhat efficiently in practice, despite their known worst-case exponential time complexity. In the quest for developing such decision procedures, several different approaches have been investigated in the last two decades, ranging from solutions that try to improve/optimise on the choice of high-level algorithm to reason about parity games, the programming language used to implement such a solution, the concrete data structures used to represent the games, or the type of hardware architecture used for deployment [7,6,17,9].

Progress solving parity games in practice has been made in different directions. In [7], a state-of-the-art implementation of the best-known algorithms for solving parity games was presented. In this work, two algorithms were found to deliver the best performance in practice, namely, Zielonka's recursive algorithm (ZRA [18]) and priority promotion [3], with the former showing slightly better performance when solving random games and a selection of structured games for model checking, and the latter outperforming ZRA when solving a selection of structured games for equivalence checking. But, overall, the two algorithms expose extremely similar performance in practice, including that of a parallel implementation of ZRA. Another attempt to improve the performance of solving parity games is presented in [6]. In this work, better performance is sought through a parallel implementation of ZRA, known to consistently expose the best performance in different platforms and for different types of games.

These two works [7,6] contain two strikingly opposing conclusions. While in [7] the parallel implementation of ZRA is even outperformed by the best sequential implementation of the same algorithm, in [6] significant gains in performance are observed when parallelising the computation of ZRA – which may solve a large set of random parity games between 3.5 and 4 times faster than the sequential implementation of the same algorithm. These two results, arguably, both conforming with the state of the art in the solution of parity games in practice, indicate that no definitive conclusion can be made into what the best approach to solving parity games in practice is, let alone whether considering a parallel implementation would necessarily produce better results than its sequential version. In this paper, we present a new approach to solving parity games, and investigate some of the issues exposed by the two above papers.

More specifically, motivated by the need to find effective new techniques for solving parity games, in particular in large practical settings, in this paper we:

1. propose a novel matrix-based approach to solving parity games, based on ZRA [18,13], arguably, the best-performing algorithm in practice [7];
2. study the complexity of our matrix-based procedure, and show that it retains the optimal complexity bounds of the best algorithms for parity games [13];
3. develop a parallel implementation, which takes advantage of methods and hardware for matrix manipulation using sophisticated GPU technologies;
4. investigate a number of alternative implementations of our matrix-based approach in order to better assess its usefulness in practical settings.

Our matrix-based approach, whose parallel implementation outperforms the state-of-the-art solvers for parity games, consists in the reduction of key operations on parity games as simple computations on large matrices, which can be significantly accelerated in practice using sophisticated techniques for matrix manipulation, specifically, using modern GPU technologies. Firstly, our matrix-based approach partly builds on the observation that most of the computation time when using ZRA is spent running a particular subroutine called the "attractor" function, which we can parallelise. Secondly, we also rely on the observation that computations on matrices – which guide the search for the solution of parity games within our approach – can be efficiently parallelised using a combination of both algorithmic techniques for parallel computation and GPU devices.

2 Preliminaries

A parity game is two-player zero-sum infinite-duration game played over a finite directed graph $G = (V_0, V_1, E, \Omega)$, where $V = V_0 \cup V_1$ is a set of vertices/nodes partitioned into vertices V_0 controlled by Player Even/0 and vertices V_1 controlled by Player Odd/1. Whenever a statement about both players is made, we may use the letter q ($\in \{0, 1\}$) to refer to either player, and $1 - q$ to refer to the other player in the game. Without any loss of generality, we also assume that every vertex in the graph has at least one successor. Moreover, the function $\Omega : V \to \mathrm{N}$ is a labelling function on the set of vertices of the graph which assigns each vertex a *priority*. Intuitively, the way a parity game is played is by moving a token along the graph (starting from some designated node in V), with the owner of the node of which the token is on selecting a successor node in the graph. Because every vertex has a successor, this process continues indefinitely, producing a infinite sequence of visited nodes, and consequently an infinite sequence of seen priorities. The winner of a particular play is determined by the highest priority that occurs infinitely often: Player 0 wins if the highest infinitely recurring priority is even, while Player 1 wins if the highest infinitely recurring priority is odd. Parity games are determined, which means that it always the case that one of the two players has a strategy (called a winning strategy) that wins against all possible strategies of the other player. Solving a parity game amounts to deciding, for every node in the game, which player has a winning strategy for the game starting in such a node. That is computing disjoint sets $W_0 \subseteq V$ and $W_1 \subseteq V$ such that Player q has a winning strategy to win every play in the game that starts from a node in W_q, with $q \in \{0, 1\}$.

Somewhat surprisingly, the best performing algorithm to solve parity games in practice is Zielonka's Recursive Algorithm (ZRA [18]), which runs in exponential time in the number of priorities, bounded by $|V|$. This algorithm is rather simple, and mostly relies on the computation of *attractor* sets, which are sets of vertices $A = Attr_q(X)$ inductively defined for each Player q as shown below – and used to computing both W_0 and W_1 recursively. Formally, the attractor function $Attr_q : \mathcal{P}(V) \to \mathcal{P}(V)$ for Player q, computes the attractor set of a given set of vertices $U \subseteq V$, and is defined inductively as follows:

Algorithm 1 *Zielonka*(G)

if $V = \emptyset$ **then**
  $(W_0, W_1) \leftarrow (\emptyset, \emptyset)$
else
  $m \leftarrow \max\{\Omega(v) \mid v \in V\}$
  $q \leftarrow m \bmod 2$
  $U \leftarrow \{v \in V \mid \Omega(v) = m\}$
  $A \leftarrow Attr_q(U)$
  $(W'_0, W'_1) \leftarrow Zielonka(G \setminus A)$
  if $W'_{1-q} = \emptyset$ **then**
    $(W_q, W_{1-q}) \leftarrow (A \cup W'_q, \emptyset)$
  else
    $B \leftarrow Attr_{1-q}(W'_{1-q})$
    $(W'_0, W'_1) \leftarrow Zielonka(G \setminus B)$
    $(W_q, W_{1-q}) \leftarrow (W'_q, W'_{1-q} \cup B)$
  end if
end if
return (W_0, W_1)

$$
\begin{aligned}
Attr_q^0(U) &= U \\
Attr_q^{n+1}(U) &= Attr_q^n(U) \\
&\quad \cup \{u \in V_q \mid \exists v \in Attr_q^n(U) : (u, v) \in E\} \\
&\quad \cup \{u \in V_{1-q} \mid \forall v \in V : (u, v) \in E \Rightarrow v \in Attr_q^n(U)\} \\
Attr_q(U) &= Attr_q^{|V|}(U)
\end{aligned}
$$

As shown in Algorithm 1, ZRA [18] finds disjoint sets of vertices W_0/W_1 from which Player $0/1$ has a winning strategy. Through the computation of attractor sets, the algorithm works by recursively decomposing the graph, finding sets of nodes that could be forced towards the highest priority node(s), and hence building the winning regions W_0 and W_1 for each player in the game.

3 A matrix-based approach

Experimental results from [7] motivated us to investigate whether ZRA can be improved in practice, since such an algorithm shows the best performance both in random games as well as in several structured games found in practical settings. This finding is complemented by the observation made in [6], that when running ZRA most of the time is spent in the computation of attractor sets, reported to be about 99% in [6] (with experiments considering random games only), and found to be of about 77% in our study (which considers larger classes of games).

Our observation, and working hypothesis, not found in previous work [7,6], is that the basic ZRA can be highly optimised in practice if its main computation component – the attractor set subroutine – is accelerated using efficient

Algorithm 2 $Attr(A, \mathbf{t}, q, \mathbf{g}, \mathbf{o})$

$\mathbf{d} \leftarrow A\mathbf{g}$
$\mathbf{t}' \leftarrow \mathbf{0}$
while $\|\mathbf{t} \neq \mathbf{t}'\|_1 \neq 0$ **do**
  $\mathbf{t}' \leftarrow \mathbf{t}$
  $\mathbf{v} \leftarrow A\mathbf{t}$
  $\mathbf{t} \leftarrow \mathbf{g} \odot ((\mathbf{o} = q) \odot (\mathbf{v} > 0) + (\mathbf{o} = (1 - q)) \odot (\mathbf{v} = \mathbf{d}))$
end while
return t

techniques for matrix manipulation, should a representation of the attractor set procedure was based on computations/operations on matrices encoding the attractor set subroutine in ZRA. This is precisely what we do in this section, which in turn makes our approach incredibly appropriate for an implementation in parallel using modern GPUs technologies for efficient matrix manipulation.

To achieve a matrix-based encoding of ZRA, and in particular of its attractor set subroutine, we redefine the representation of the graph in terms of a sparse adjacency matrix A, a vector defining the ownership of every node $\mathbf{o}$, and a vector $\boldsymbol{\omega}$ defining the priority of every node. Due to the potentially high computational cost of copying A, we maintain a vector $\mathbf{g}$ representing which nodes are still included in the game (a subgame being computed at that point in the algorithm), which is copied and updated as Zielonka's algorithm recurses and decomposes the graph into ever smaller parts. As such, we are able to find $\mathbf{d} = A\mathbf{g}$, a vector containing the maximum out-degree of every node. More specifically:

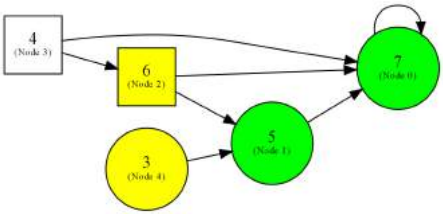
- $(A)_{ij} = 1$, if edge exists connecting i and j; $(A)_{ij} = 0$, otherwise;
- $(\mathbf{o})_i = q$, if node i belongs to player q;
- $(\boldsymbol{\omega})_i = \Omega(V_i)$;
- $(\mathbf{g})_i = 1$, if node i is in the game; $(\mathbf{g})_i = 0$, otherwise.

With these definitions in place, we can make the necessary modifications to the attractor function presented before – see Algorithm 2. The input/output vector $\mathbf{t}$ contains 1 at position $(\mathbf{t})_i$ where a node i is part of the attractor set and 0 otherwise. We thus define vectorised operations where if a vector is compared to another vector, then the comparisons are done element-wise. If a vector is compared to a scalar, then the scalar s is implicitly converted, $\mathbf{s} = s\mathbf{1}$. The $\odot$ operator denotes the Hadamard product, which is used primarily as a Boolean **And** operation. The argument q is the player: 0 for Player 0 and 1 for Player 1.

This algorithm works by first finding the number of outbound edges each node has ($\mathbf{d} \leftarrow A\mathbf{g}$), and at each iteration finding how many ways each node can enter the attractor set ($\mathbf{v} \leftarrow A\mathbf{t}$). It then finds nodes that q owns that may enter the attractor set ($(\mathbf{o} = q) \odot (\mathbf{v} > 0)$), and nodes that q do not own that are forced to enter the attractor set ($(\mathbf{o} = (1 - q)) \odot (\mathbf{v} = \mathbf{d})$). It then filters the nodes to include into the attractor set depending on which nodes are still included in the subgraph ($\mathbf{g} \odot (\cdots)$), and breaks the loop when there is no difference between $\mathbf{t}$ and $\mathbf{t}'$. To illustrate this procedure, take as an example the graph below.

Algorithm 3 $MatZielonka(A, \mathbf{g}, \mathbf{o})$

if $\|\mathbf{g}\|_1 = 0$ **then**
  $(W_0, W_1) \leftarrow (\mathbf{0}, \mathbf{0})$
else
  $m \leftarrow \max(\mathbf{g} \odot \boldsymbol{\omega})$
  $q \leftarrow m \mod 2$
  $\mathbf{t} \leftarrow (\boldsymbol{\omega} = m)$
  $\mathbf{t} \leftarrow Attr(A, \mathbf{t}, q, \mathbf{g}, \mathbf{o})$
  $(W'_0, W'_1) \leftarrow MatZielonka(A, \mathbf{g} - \mathbf{t}, \mathbf{o})$
  if $\|W'_{1-q}\|_1 = 0$ **then**
    $(W_q, W_{1-q}) \leftarrow (\mathbf{t} + W'_q, \mathbf{0})$
  else
    $\mathbf{t} \leftarrow Attr(A, W'_{1-q}, 1-q, \mathbf{g}, \mathbf{o})$
    $(W'_0, W'_1) \leftarrow MatZielonka(A, \mathbf{g} - \mathbf{t}, \mathbf{o})$
    $(W_p, W_{1-p}) \leftarrow (W'_q, W'_{1-q} + \mathbf{t})$
  end if
end if
return (W_0, W_1)



$$A = \begin{pmatrix} 1&0&0&0&0 \\ 1&0&0&0&0 \\ 1&1&0&0&0 \\ 1&0&1&0&0 \\ 0&1&0&0&0 \end{pmatrix}$$

For this example, assume that $\mathbf{g} = \mathbf{1}$ and that we are computing the attractor set for the player that own the circle nodes, starting from the node with priority 7. After 1 (or some arbitrary number of iteration(s)), the current state is reached. Green nodes denote nodes included in the previous iteration's attractor set, and yellow nodes denote nodes that will be included in this iteration. The calculations that may be performed are as follows. Define the adjacency matrix of the graph (A), the currently included nodes in the attractor set, $\mathbf{t} = (1\ 1\ 0\ 0\ 0)^\top$, the ownership of every node, $\mathbf{o} = (0\ 0\ 1\ 1\ 0)^\top$, and the degree – number of outbound edges – of every node, $\mathbf{d} = A\mathbf{g} = (1\ 1\ 2\ 2\ 1)^\top$. Now, compute the number of edges from each node leading to an element in the current attractor set, that is, $\mathbf{v} = A\mathbf{t} = (1\ 1\ 2\ 1\ 1)^\top$, and with that, update $\mathbf{t}$, to obtain: $\mathbf{t} \leftarrow (1\ 1\ 1\ 0\ 1)^\top$, which exactly represents the value of the attractor function one step later. Similar changes for ZRA in terms of the representation of the game must also be made, so that it becomes, fully, a matrix manipulation algorithm (Algorithm 3).

The correctness of the algorithm remains unchanged from that of ZRA since our encoding into matrix operations is functional. Less clear is whether our algorithm retains the ZRA's complexity, since using a functional mapping does not necessarily imply that the encoding (our representation) has the complexity of the encoded instance (*i.e.*, the original problem). We study this question next.

3.1 Complexity

Using the algorithms defined before, we derive a function $R(d, n)$ that bounds the maximum number of recursive calls to ZRA, given a d number of distinct priorities and n nodes: $R(d, n) = 1 + R(d-1, n-1) + R(d, n-1)$. The 1 is the original call; the 1st recursive call is made with at least the vertex with the largest priority removed, and the second is made with at least one vertex removed. Hence, the construction above. There are two base cases $R(d, 0) = R(0, n) = 1$. Firstly, we observer that based on the algorithms herein defined, we get:

$$\begin{aligned} R(d, n) &= 1 + R(d-1, n-1) + R(d, n-1) \\ &= (n+1) + \sum_{i=1}^{n} [R(d-1, n-i)] \end{aligned}$$

Moreover, $R(d, n)$ is then given by: $f(d, n) = 2\sum_{j=0}^{d} \binom{n}{j} - 1$. For the base case, when $d = 1$, we note that $R(1, n) = (n+1) + \sum_{i=1}^{n} [R(0, n-i)] = 2n + 1$ and $f(1, n) = 2\sum_{j=0}^{1} \binom{n}{j} - 1 = 2(n+1) - 1 = 2n + 1 = R(1, n)$, as required, for all n. For the inductive case, assume that $R(d, n) = f(d, n)$, for $d = k$ and all n.

$$\begin{aligned} R(k+1, n) &= (n+1) + \sum_{i=1}^{n} [R(k, n-i)] \\ &= (n+1) + \sum_{i=1}^{n} [f(k, n-i)] \\ &= 1 + 2\sum_{i=1}^{n}\sum_{j=0}^{k} \binom{n-i}{j} = 2\sum_{j=0}^{k+1} \binom{n}{j} - 1 = f(k+1, n) \end{aligned}$$

Hence, the statement is true for the base case $d = 1$ and all n, while the inductive case $d = k$ implies $d = k + 1$. Thus, by induction, $R(d, n) = f(d, n)$ for $d \geq 1$ and all n. We now observe that the worst case number of calls occurs, as expected, at $d = n$ where $R(n, n) = 2^{n+1} - 1$. Note that the complexity of a single call to *MatZielonka* has time complexity $O(n^3)$ (dominated by the complexity of calls to the matrix-based *Attr* subroutine[1]) and space complexity $O(n)$, delivering worst-case complexities of $O(n^3 \cdot 2^n)$ time and $O(n \cdot 2^n)$ space.

This result, negative in theory, is consistent with that of the worst-case complexity of ZRA, which indicates that our matrix-based encoding retains the same complexity properties of the original algorithm. More interestingly, is the fact that the quasi-polynomial extension of ZRA by Parys [16], and later improved by Lehtinen et al [13], can also be tackled with our approach while retaining the quasi-polynomial complexity. However, a matrix-based extension of the latter algorithm was not evaluated. Thus, its practical usefulness is yet to be studied.

[1] In practice, this is dominated by the complexity of performing matrix multiplication operations, which is just slightly larger than $O(n^2)$ and happens to be a vibrant topic of research recently due to improvements made through the use of Deep learning.

4 Implementation and evaluation

Several factors influence the practical performance of a computational solution to a problem: for instance, (1) the algorithm used to solve the problem, (2) the programming language to implement the solution, (3) the concrete data structures used to represent it, and (4) the hardware where the solution is deployed. Our solution tries to optimise 1–4 using both lessons learnt from previous research and properties of our own matrix-based approach. Details are given later, but in short, in this section, five parity game solvers are implemented and evaluated[2]:

I1 our basic matrix-based approach, presented in the previous section;
I2 its parallel implementation for deployment using GPU technologies;
I3 the improved implementation of the attractor function of ZRA in [6];
I4 the highly optimised C++ implementation of ZRA presented in [7];
I5 the unoptimised version of the above algorithm, also in [7].

Apart from (2), the five implementations above (I1–I5) will allow you to have a comprehensive evaluation of our approach, both against different versions of our own work and against previous research. The only aspect that all the solutions we present in this section have in common is the programming language used for implementation, which is C++, at present the language offering the most efficient practical implementation of parity games solutions; cf. [9,17,6,7]. We first present the characteristics of our matrix-based approach, deployed both as a sequential algorithm and as a parallelised procedure. After that, we will describe key features of the solutions originally developed elsewhere, and continue with the results of the evaluation using different types of parity games.

Matrix-based approach.[3] Whilst it is important to find performance from parallelisable operations, it is equally important to avoid the loss of performance from executing inefficient or slow operations. Specific algorithmic design choices such as maintaining a vector $\mathbf{g}$ to track nodes that are in or out of the graph are done to avoid otherwise necessary operations such as copying the adjacency matrix, which would otherwise be slow, especially when solving very large games.

Additionally, all values in vectors and matrices are stored as single precision floating point values in practice. This is due to the software limitations of the Compute Unified Device Architecture (CUDA) [15] library, which are likely limitations of the underlying hardware itself. In particular, this limits the maximum out-degree of a node to 2^{24}, which corresponds to the number of bits in the mantissa of a single precision floating point number (23), plus one. Beyond this limit, the accuracy of the values computed in operations such as computing the maximum out-degree of a node with $A\mathbf{g}$ would no longer be guaranteed, along with the correctness of the algorithm. We note that this limitation may be overcome by splitting a single node into multiple nodes, thus curbing the maximum out degree to an acceptable range. We do not do this for these experiments as this transformation has unknown impacts on the performance of the algorithm.

[2] All files (implementations, experiments, input games, etc.) can be found in [1].
[3] The description here applies to the first two solutions described above.

Algorithm 4 $Attr(A, \mathbf{t}, q, \mathbf{g}, \mathbf{o})$

```
...
while ‖t ≠ t'‖_1 ≠ 0 do
    for i ∈ (1..3) do
        ...
    end for
end while
return t
```

The invocation of functions that run on the GPU (known as *kernels*) have an overhead, with the overhead duration varying somewhat between devices. As a consequence, tuning for a particular problem depends on the functions being executed and the GPUs themselves. Thus, there are periods where the device is idle, and this is a result of the overheads. Also note that in practice, it is usually faster to perform multiple iterations of the attractor computation as performing an iteration when the full attractor set has already been computed does not alter the results (Algorithm 4). This is because queueing multiple kernel invocations has the same overhead as calling one kernel alone. The main difference between our sequential and parallel implementations of the matrix-based method is the function computing attractor sets, which is as in Algorithm 2 in the sequential case, and as in Algorithm 4 in the parallel case. The code in ... is the same in both implementations, and the key difference is that we set the execution of the parallel implementation to make 3 kernel invocations per execution of the attractor function – which in lucky cases may require only 1 kernel invocation, while in unlucky cases may require more than 3 kernel invocations, increasing overheads; for our problem, we found that 3 kernel invocations was appropriate.

We find that there is another possible point of optimisation as the time taken for the attractor computation would be approximately equal to $ct_c + nt_o$, where c is the number of attractor computations (the inside section of the for loop), n is the number of times the outer while loop will run, t_c is the time to run the for loop once, and t_o is the overhead incurred by switching execution from device (GPU) to host (CPU) as the condition is checked in the while loop. Ideally, $c = C+1$, and $n = 1$, where C is the (unknown) number of attractor computations required. Our implementation loops the inner for loop an arbitrary constant number of times (3 times here). As such, $C + 1 \leq c \leq C + 3$, and $n = \lceil \frac{C}{3} \rceil$.

Importantly, requirements for the efficient parallelisation of the algorithm on the GPU require us to select the 'Naive attractor' implementation as the underlying algorithm (Algorithm 2) to be parallelised (leading to Algorithm 4) rather than the 'Improved attractor' implementation in [6]. The concepts of 'Naive' and 'Improved' attractors are presented by Arcucci et al in [6]. In short, the 'Naive' attractor loops over each node and checks if it can be included in the attractor set, and repeats this until no further nodes can be added. The 'Improved' attractor starts from the original attractor set, performing backpropagation on their inbound edges to find other nodes that may be included in the set.

GPU deployment. Our GPU implementation works by parallelising the "attract" operation.[4] Whilst the sequential version may be executed as such:

- (Loop 1) While attracting new nodes...
  - (Loop 2) For each node, check if it can be included in the attractor set.

And the runtime operations may look like:

- While attracting new nodes...
  - Can node 1 be included in the attractor set?
  - ...
  - Can node N be included in the attractor set?
- If attracted new nodes, repeat loop. Else break.

Performance is found through the inner loop being efficiently parallelised on the GPU. Additional specifics include the following GPU deployment features. When asking "Can node X be included ...?", the computation taking place is:

- Let J be the set of nodes in the current attractor set.
- Let K be the set of nodes that X can move to.
- If X is on the "friendly" team, and $K \cap J \neq \emptyset$, then $J \leftarrow J \cup \{X\}$.
- If X is on the "enemy" team, and $K \subseteq J$, then $J \leftarrow J \cup \{X\}$.

Key to our approach is that these operations are efficiently parallelised through means of matrix multiplication operations on the GPU. It is done as such:

- Compute $\mathbf{t} = A\mathbf{1}$. Hence, t_i is the number of nodes node i can move to.
- Let $\mathbf{j}$ be a vector of size N (where N is the size of the parity game), such that $\mathbf{j}_i = 1$ if and only if node i is in the current attractor set. Default 0.
- Let A be an adjacency matrix (usually, a sparse matrix) of the parity game.
- Compute the vector $\mathbf{k} = A\mathbf{j}$. Hence, the value $\mathbf{k}_i$ in the vector is the number of nodes node i can move to and that are in the current attractor set.
- Then, for each node i, if it is on the friendly team, and $\mathbf{k}_i \neq 0$, then $\mathbf{j}_i = 1$; otherwise, if it is on the enemy team, and $\mathbf{k}_i = \mathbf{t}_i$, then $\mathbf{j}_i = 1$.

Note we convert the previous logic on sets to suit the new form using vectors:

$K \cap J \neq \emptyset \Leftrightarrow \mathbf{k}_i \neq 0$ and $K \subseteq J \Leftrightarrow \mathbf{k}_i = \mathbf{t}_i$.

Improved attractor implementation by Arucci et al [6]. The third parity game solver we evaluate is a custom, C++, implementation of the ZRA using the 'Improved attractor' algorithm in [6], originally implemented in JAVA there.

ZRA implementations in Oink [7]. The fourth and fifth implementations we evaluate and compare against are the most highly optimised implementation of ZRA developed in [7], and its unoptimised version – without pre-processing routines. We include this implementation since our matrix-based ('Naive') implementation is not optimised in terms of the pre-processing routines used for implementation. These solvers in Oink are referred to as `zlk` and `uzlk` in [7]. We note that the parallel implementation of this algorithm is not included since in [7] is shown that it usually is outperformed by `zlk`, which we include here.

[4] A very different approach, leading to a very different GPU deployment is done in [10].

4.1 Evaluation

The implementations evaluated in this paper were tested on a wide repository of parity games, and against state-of-the-art parity game solvers in the literature. The games used for performance evaluation include the suite by Keiren [12] (of games representing model checking and equivalence checking problems) and an additional set of variably sized random games generated by PGSolver [9].[5]

We evaluate the performance in terms of solve time of each of the solvers and for each of the games. As it is common practice when evaluating different solvers for parity games, the overheads incurred due to startup and game loading are not included; this is done in order to obtain numbers that estimate only the running time of the algorithms, and nothing else. With the same aim, we ensured that at most one solver is running at any time, with CPU utilisation not exceeding more than one core. Finally, in order to allow for a fair comparison of running times *only* – rather than combining such results with the robustness of the algorithms – we measured the time solving an instance only in case all implementations successfully compute a solution. This allows for a fairer comparison with respect to runtime performance purely, because failing a game usually implies an extremely disproportionately (and arbitrary) high runtime. Such failures include timeouts (at 5 minutes) or being unable to load the game, sometimes due to factors having little to do with the running time of the algorithms. Our experiments were conducted in the Google Cloud Platform (GCP) using a T4 n1-highmem-2.[6]

Profile of the input parity games. Our study includes more than 2000 parity games, with sizes ranging from only a few dozens of states to games with millions of states. Both nodes' out-degrees and number of distinct priorities also cover a wide range of dimensions. However, both random games and structured ones (model checking and equivalence checking) typically are represented by sparse graphs, a feature that we will leverage for implementation purposes.

5 Analysis of results

As can be seen from Tables 1, 2, and 3, we evaluate the main five implementations, all of them following the ZRA philosophy, using two types of parity games: structured and random. Both types of benchmarks are as in [7] and [6], arguably, the two best implementations of ZRA. The focus of this evaluation is to understand the usefulness and scalability of the 'GPU matrix' algorithm, which is the one embodying more cleanly our working hypothesis, namely, that the combination of a matrix-based representation of ZRA and the use of modern GPU technologies can outperform the state of the art in the design of algorithms for parity games – a hypothesis for which we provide strong evidence here.

[5] These random games were generated using parameters that are identical to those of the random games in the 'PGSolver' collection in the suit of benchmarks by Keiren.

[6] In order to compare performance in different hardware (GPU) architectures, we use a different technology for experiments presented in a forthcoming section.

	Model checking		Equiv checking		Random games	
Implementation	Time	P/F	Time	P/F	Time	P/F
GPU matrix	**94**	313/0	**332**	209/7	**20**	1750/0
Naive (matrix) attractor	566	313/0	2190	216/0	88	1750/0
Improved attractor	212	313/0	1310	216/0	113	1750/0
Oink's `zlk`	143	313/0	578	216/0	39	1750/0
Oink's `uzlk`	150	313/0	917	216/0	69	1750/0

Table 1: Times are in milliseconds (ms) representing the average time taken to solve games that all implementations passed (*i.e.*, if *any* implementation fails to solve a game, the game is excluded from the time average of *all* five solvers, including an additional GPU implementation on an RTX2060S, presented later). Failures occur with a small number of large equivalence checking games only. Failures include a few timeouts (at 5 mins), and usually being unable to load the game in memory due to hardware limitations posed by the GPU architectures. Columns P/F show the number of games passed/failed for every type of game.

	Model checking		Equiv checking		Random games	
Implementation	Time	P/F	Time	P/F	Time	P/F
GPU matrix	**814**	33/0	**2612**	29/7	**283**	50/0
Naive (matrix) attractor	4565	33/0	17610	36/0	1059	50/0
Improved attractor	1832	33/0	10411	36/0	1446	50/0
Oink's `zlk`	1263	33/0	4568	36/0	547	50/0
Oink's `uzlk`	1316	33/0	7332	36/0	952	50/0

Table 2: Results in this table are formatted as in Table 1. In this table, we report the performance (average time in milliseconds taken to solve a single game) for the 5 algorithms on large (>1M nodes) parity games only.

	Model checking		Equiv checking		Random games	
Implementation	Time	P/F	Time	P/F	Time	P/F
GPU matrix	**9**	280/0	**22**	180/0	**12**	1700/0
Naive (matrix) attractor	95	280/0	172	180/0	59	1700/0
Improved attractor	21	280/0	119	180/0	74	1700/0
Oink's `zlk`	11	280/0	56	180/0	24	1700/0
Oink's `uzlk`	13	280/0	77	180/0	43	1700/0

	Nodes								
Implementation	4K	8K	12K	16K	20K	40K	80K	320K	640K
GPU matrix	1	1	2	**2**	**2**	**5**	**11**	**43**	**78**
Naive (matrix) attractor	1	1	2	4	5	17	37	208	469
Improved attractor	1	1	3	5	7	19	45	264	557
Oink's `zlk`	1	1	2	3	4	7	15	76	186
Oink's `uzlk`	1	2	3	4	5	11	25	142	354

Table 3: Results in this table are formatted as in Table 1. In this table, we report the performance (average time in milliseconds taken to solve a single game) for the 5 algorithms on "small" (<1M nodes) parity games only: results for structured and random games appear in the top table and for random games (detailed) at the bottom. In the bottom table, there are 200 games per column, apart from column 640K which has 100 games; there are no failures.

The results above also show that going from the sequential version of our approach, 'Naive (matrix) attractor' to its parallel implementation using GPU technologies finds significant improvements. These two main "internal" results are then compared with the state of the art in the algorithmic design of solutions based on ZRA, namely, using the improved attractor in [6] and using the highly optimised procedure `zlk` in Oink [7], which even outperforms its own parallel implementation; cf. [7]. Finally, the unoptimised version in Oink of this procedure, `uzlk`, is also included simply because our matrix-based procedure does not contain any of the pre-processing routines that differentiate `zlk` from `uzlk`. Thus, in a way, `uzlk` provides results for a somewhat fairer comparison.

GPU matrix vs Naive (matrix) attractor. Results in all tables show that the parallel implementation using GPU technologies outperforms its own sequential implementation ('Naive matrix attractor') by several orders of magnitude, with some exceptions, usually ranging from 5 times faster in some cases (e.g., model checking of large games) to more than 10 times faster (e.g., model checking of small games). This, we believe, is due to the fact that the bigger the input instances to be analysed the more any losses in the associated overheads of running the procedure in parallel are compensated later on. A trend going in that direction can be observed in detail when comparing the performance of these two algorithms over small random games. But, in any case, our matrix-based approach is always at least as good as its sequential implementation.

GPU matrix vs Improved attractor. The results show that the parallel matrix-based approach can outperform the improved attractor procedure by Arcucci et al [6] by 2-7 orders of magnitude, depending on the type of game being solved, and with the best results obtained when solving random games, whether large or small. However, the sequential version of 'GPU matrix', that is, the Naive implementation, usually is twice slower than the improved attractor implementation in structured games. Contrarily, even the (sequential) Naive implementation of the matrix-based method outperforms the improved attractor procedure over random games, being about 30% overall in that case. When looking at all the tables of results together, one can see that this is in fact an indicator of the fact that the improved attractor approach performs somewhat poorly over random graphs, at least when compared to its performance over structured games.

GPU matrix vs Oink. Even thought the GPU matrix-based implementation outperforms Oink's `zlk`, it usually does it only by a 1.5 to 2.0 factor, with the GPU implementation performing more efficiently over (large) random games than over structured ones. This result actually speaks very highly of the optimised sequential implementation of ZRA. However, as shown in [7], `zlk` performs even better than its own parallel implementation (called `zlk-8` in [7]) when solving model checking parity games (by a very small margin) and when solving random games, where it is nearly twice faster; cf. Table 3 of [7]. Only when solving equivalence checking parity games `zlk-8` outperforms `zlk`, but only by about a 13% margin.

In contrast, the GPU implementation here outperforms `zlk` by more than a 70% margin, and is even twice faster when solving small equivalence checking games.

However, as we can see from all tables, the GPU matrix-based implementation has some failures (running timeout or failure to upload the game in memory, mainly due to their size), while the improved attractor method never fails in the considered set of benchmarks. This indicates that in this particular case, there may be a choice to be made between some potentially marginal gain in efficiency and more reliability offered by `zlk`. On the other hand, `zlk` clearly outperforms the sequential (Naive) implementation of the matrix-based approach, with better efficiency going from twice faster when solving random games to about four times faster when solving structured games. Regarding performance against Oink's `uzlk`, all analyses above remain similar, only that a better factor is usually obtained in favour of the GPU matrix-based approach.

Improved attractor vs Oink's `zlk`. Despite these two procedures being originally developed previously, we would like to comment on their comparative performance, for the sake of completeness of the analysis. As can be seen from our results, both offer the same reliability as they do not fail to solve any instance. Regarding runtime efficiency, we can observe that, on average, Oink's `zlk` implementation tends to be 1.5 to 3.0 times faster than the improved attractor method, with the worst/best comparative performance being enacted when solving model checking/random parity game instances, and in that way making `zlk` perhaps the most efficient sequential implementation of ZRA currently available in the literature, and being outperformed only when a parallel approach is considered.

6 Special cases

In this section, we analyse in more details two special cases of our results: performance when solving large parity games and performance on random games.

6.1 Solving large parity games

For the purposes of this section, a *large* parity game is a game with more than 1 million nodes. Our results show that for games that are not large (Table 3), all solvers may be regarded as running efficiently from a human perspective, with some random games with more than 500K nodes being solved in about half a second by the slowest implementation on random games (the improved attractor implementation). In most other instances, solutions may be obtained in just a few milliseconds. For instance, model checking parity games in the suite of benchmarks can be solved in less than 0.1 minutes by any studied solver, and even in less than 10 milliseconds on average using the parallel GPU matrix-based approach, with Oink implementation taking virtually the same time (just a little more than 10 milliseconds on average). Then, the real challenge when solving parity games in practice is solving large parity games, where the relative performances between different solvers can be much better exposed (Table 2).

Our results show (Tables 1 and 2) that, despite the raw data being different in about 9 orders of magnitude, nearly the same relative performance is obtained when looking at performance over all games with respect to performance over large games only, which account for no more than 15% of the games for equivalence checking games, 10% for model checking games and less than 5% for random games. This result indicates that in order to evaluate the performance of parity games solvers in practice, one should better focus on large games only. As the data shows, in that case that parallel GPU matrix-based approach outperforms the second-best technique by, approx., a 1.5-2.0 factor, and its own sequential implementation by a factor of 4 to 5, in each case, depending on the type of parity game under consideration. The analysis holds across all solvers.

6.2 Solving random parity games

Random parity games are a common benchmark for parity games solvers, being the focus of the study on [6]. Our detailed experiments on random parity games show that the parallel GPU implementation of the matrix-based approach is comparable to the parallel implementation of the improved attractor implementation in [6] (see Table 3 there), in the sense that a similar relative *gain* in performance is achieved, overall, performing about 3.5-4.0 times faster over random games of up to 20K nodes. The gain in performance increases in our case when considering larger random graphs, perhaps indicating that our approach may be more scalable in terms of running time; however, in [6], only results on random games of up to 20K nodes are presented. We note that, in this case, only by changing the programming language of choice (JAVA in [6] and C++ here), performance is improved going from games of 20K size being solved in more than 5 seconds to the same type of games being solved in just 7ms on average here.

7 Alternative implementations

In this section, we explore two alternative implementations, one focused on a change of programming environment and another one based on a change of computer architecture. Our results show that while the former is well outperformed by the original C++ implementation, the latter shows even better performance than the already reported can be achieved when using other GPU technologies.

A MATLAB *implementation.* Given its facility to perform matrix operations, we investigated a MATLAB of our matrix-based approach to understand if it could perform better than our original C++ implementation. The results were negative. The MATLAB implementation of our approach, although simple, performed significantly worse than other methods, including our own using C++. A summary of the results, which require little discussion, can be found in Table 4.

Using a different GPU technology. We conducted experiments using the exact same implementation of the GPU matrix solver (run on a GCP) on a different

	Model checking		Equiv checking		Random games	
Implementation	Time	P/F	Time	P/F	Time	P/F
GPU matrix	**94**	313/0	**332**	209/7	**20**	1750/0
Naive (matrix) attractor	566	313/0	2190	216/0	88	1750/0
MATLAB matrix	2462	311/2	2338	198/18	3496	1750/0

Table 4: Results in this table are formatted as in Table 1. We report results on all games, and in each case, independently, remove the time of unsolved instances.

	Model checking		Equiv checking		Random games	
Implementation	Time	P/F	Time	P/F	Time	P/F
GPU matrix (RTX2060S)	**63**	313/0	**203**	205/11	25	1750/0
GPU matrix	94	313/0	332	209/7	**20**	1750/0

Table 5: Results in this table are formatted as in Table 1. We report results on all games, which show an improvement of a 1.5x factor for structured games, while performing approximately 25% slower over random parity games.

GPU architecture, namely, on an RTX2060 Super (Ryzen 5 3600). We found that by simply changing to this alternative hardware specification, the results on all types of games were significantly better, as shown in the Table 5.

8 Concluding remarks and related work

We have shown that a new method for solving parity games using a matrix-based approach can outperform the state-of-the-art techniques, both sequential and parallel, currently available. As such, our results become a new point of comparison when evaluating modern solvers for parity games. Previous research [7,6,17,9] has shown that ZRA is potentially the best performing algorithm to solve parity games in practice, and here we provide more evidence that this is indeed the case. We also give evidence that C++ implementations for this task are hardly ever outperformed in practice. Finally, we also show that choosing the right computer architecture is key to achieve optimal performance, and in particular that in the case of modern GPU technologies, such a choice can make a significant difference in practice – in our study, leading to the development of the, as of today, most efficient parallel implementation/solver for parity games.

Acknowledgement. This research was funded by the Monash Laboratory for the Foundations of Computing (MLFC) and the Monash Faculty of Information Technology (FIT). Parts of this research were developed as FIT3144 projects ("Advanced computer science research project") at Monash in 2022. Preliminary results on the matrix-based approach to parity games were also developed by Henri Urpani during his FIT3144 project under Gutierrez's supervision in 2021. Finally, we thank the reviewers for helpful comments that improved this paper.

References

1. Aggarwal, S., Stuckey de la Banda, A., Yang, L., Gutierrez, J.: Parity games benchmarks: Implemenation and experiments. https://drive.google.com/drive/folders/1z2eAGxU9jyn2ngnhM8c6f4Py4reW3toB?usp=sharing (January 2023)
2. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
3. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving parity games via priority promotion. Formal Methods Syst. Des. **52**(2), 193–226 (2018)
4. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasi-polynomial time. SIAM J. Comput. **51**(2), 17–152 (2022)
5. Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: Model checking, 2nd Edition. MIT Press (2018)
6. D'Amore, L., Murano, A., Sorrentino, L., Arcucci, R., Laccetti, G.: Toward a multilevel scalable parallel zielonka's algorithm for solving parity games. Concurr. Comput. Pract. Exp. **33**(4) (2021)
7. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018. LNCS, vol. 10805, pp. 291–308. Springer (2018)
8. Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy (extended abstract). In: 32nd Annual Symposium on Foundations of Computer Science. pp. 368–377. IEEE (1991)
9. Friedmann, O., Lange, M.: Solving parity games in practice. In: Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer (2009)
10. Hoffmann, P., Luttenberger, M.: Solving parity games on the GPU. In: Hung, D.V., Ogawa, M. (eds.) Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8172, pp. 455–459. Springer (2013)
11. Jurdzinski, M.: Deciding the winner in parity games is in UP ∩ co-UP. Inf. Process. Lett. **68**(3), 119–124 (1998)
12. Keiren, J.J.A.: Benchmarks for parity games. In: Dastani, M., Sirjani, M. (eds.) Fundamentals of Software Engineering. pp. 127–142. Springer (2015)
13. Lehtinen, K., Parys, P., Schewe, S., Wojtczak, D.: A recursive approach to solving parity games in quasipolynomial time. Log. Methods Comput. Sci. **18**(1) (2022)
14. Martin, D.: Borel determinacy. Annals of Mathematics **102**(2), 363–371 (1975)
15. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? Queue **6**(2), 40–53 (2008)
16. Parys, P.: Parity Games: Another View on Lehtinen's Algorithm. In: Fernández, M., Muscholl, A. (eds.) 28th EACSL Annual Conference on Computer Science Logic (CSL 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 152, pp. 32:1–32:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2020)
17. Stasio, A.D., Murano, A., Prignano, V., Sorrentino, L.: Solving parity games in scala. In: Lanese, I., Madelaine, E. (eds.) Formal Aspects of Component Software - 11th International Symposium, FACS 2014. LNCS, vol. 8997, pp. 145–161. Springer (2014)
18. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1-2), 135–183 (1998)

A GPU Tree Database for Many-Core Explicit State Space Exploration

Anton Wijs(✉) and Muhammad Osama

Eindhoven University of Technology, Eindhoven, The Netherlands
{a.j.wijs,o.m.m.muhammad}@tue.nl

Abstract. Various techniques have been proposed to accelerate explicit-state model checking with GPUs, but none address the compact storage of states, or if they do, at the cost of losing completeness of the checking procedure. We investigate how to implement a tree database to store states as binary trees in GPU memory. We present fine-grained parallel algorithms to find and store trees, experiment with a number of GPU-specific configurations, and propose a novel hashing technique, called Cleary-Cuckoo hashing, which enables the use of Cleary compression on GPUs. We are the first to assess the effectiveness of using a tree database, and Cleary compression, on GPUs. Experiments show processing speeds of up to 131 million states per second.

Keywords: Explicit state space exploration, finite-state machines, GPU.

1 Introduction

Major advances in computation increasingly need to be obtained via parallel software, as Moore's Law is ending [30]. In the last decade, GPUs have been successfully applied to accelerate various computations relevant for model checking, such as probability computations for probabilistic model checking [8,25,48], counter-example construction [54], state space decomposition [52], parameter synthesis for stochastic systems [12], and SAT solving [34–38,40,43,56,57]. VoxLogicA-GPU applies model checking to analyse (medical) images [9].

In the earliest work on GPU explicit state space exploration, GPUs performed part of the computation, specifically successor generation [18,19] and property checking once the state space has been generated [5]. This was promising, but the data copying between main and GPU memory and the computations on the CPU were detrimental for performance. The first tool that performed the entire exploration on a GPU was GPUexplore [33,50,51,53]. It was later extended to support LTL model checking [49]. A similar exploration engine was later proposed in [55]. An approach that applied a GPU to explore the state space of Promela models, i.e., the models for the Spin model checker [21], was presented in [6]. This was later adapted to the swarm checker Grapple [16], which can efficiently explore very large state spaces, but at the cost of losing completeness. Finally, the model checker ParaMoC for pushdown systems was presented in [46,47].

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 684–703, 2023.
https://doi.org/10.1007/978-3-031-30823-9_35

The above techniques demonstrate the potential for GPU acceleration of state space exploration and (explicit-state) model checking, being able to accelerate those procedures *tens* to *hundreds* of times, but they all have serious practical limitations. Several limit the size of state vectors to 64 bits [6, 55] or the size of transition encodings to 64 bits [46, 47]. GPUEXPLORE does not efficiently support models with variables [50, 53]. When adding variables, the amount of memory needed rapidly grows, due to the growing input model and inefficient state storage. GRAPPLE requires less memory, but uses bitstate hashing. This rules out the ability to detect that all reachable states have been explored, which is crucial to prove the absence of undesired behaviour. PARAMOC verifies pushdown systems, but does not support concurrency, and abstracts away data.

Contributions. We propose how to perform memory-efficient complete state space exploration on a GPU for concurrent *Finite-State Machines (FSMs) with data*. To make this possible, we are the first to investigate the storage of binary trees in GPU hash tables, propose new algorithms to find and store trees in a fine-grained parallel fashion, experiment with a number of GPU-specific configurations, and propose a novel hashing technique called *Cleary-Cuckoo hashing*, which enables the use of *Cleary compression* [13,15] on GPUs. To achieve this, we have to tackle the following challenges: 1) CPU-based algorithms are recursive, but GPUs are not suitable for recursion, and 2) accessing GPU global memory, in which the hash tables reside, is slow. This work marks an important step to pioneer practical GPU accelerated model checking, as it can be extended to checking functional properties of models with data, and paves the way to investigate the use of *Binary Decision Diagrams* [29] for symbolic model checking.

The structure of the paper is as follows. In Section 2, we discuss related work on GPU hash tables. Section 3 presents background information on GPU programming, and Section 4 contains an overview of the state space exploration engine. Section 5 addresses the challenges when designing a GPU tree table, and presents our new algorithms. Experimental results are given in Section 6, and in Section 7, conclusions and our future work plans are discussed.

2 Related Work

An overview of related work on GPU acceleration of model checking is given in Section 1. In the current section, we focus on *hash tables* [14] for the GPU. In explicit state space exploration, states are typically stored in a hash table. Such a table is often implemented as an array, where the elements represent the hash table *buckets*. A recent survey of GPU hash tables [31] identifies that when using integer data items and unordered insertions and queries, *Cuckoo hashing* [41] is (currently) the best option, compared to techniques such as *chaining* [3] or *robin hood hashing* [20], and the Cuckoo hashing of [1] is particularly effective. In Cuckoo hashing, collisions, i.e., situations where a data item e is hashed to an already occupied bucket, are resolved by evicting the encountered item e', storing e, and moving e' to another bucket. A fixed number of m hash functions

is used to have multiple storage options for each item. Item look-up and storage is therefore limited to m memory accesses, but can lead to chains of evictions. In [1], it is demonstrated that with four hash functions, a hash table needs around $1.25N$ buckets to store N items.[1] Recent research [4] has demonstrated that using larger buckets, spanning multiple elements, that still fit in the GPU cache line is beneficial for performance, and increases the average load factor, i.e., how much the hash table can be filled until an item cannot be inserted, to 99%. We address this in detail in Section 3. However, in [4], an older NVIDIA GPU of the VOLTA architecture was used (2017), while more recent GPUs are supposedly less susceptible to optimisations exploiting the cache line. In this work, we experimentally assess this for hash table buckets.

Besides buckets, we also consider Cuckoo hashing as used in [1, 4], but we are the first to investigate the storage of *binary trees*, and the use of Cleary compression to store more data in less space. Libraries offering GPU hash tables, such as [23], do not offer these capabilities. Furthermore, we are the first to investigate the impact of using larger buckets for binary tree storage embedded in a state space exploration engine.

The model checker GPUEXPLORE [11, 50, 53] uses multiple hash functions to store a state. State evictions are never performed, as each state is stored in a sequence of integers, making it not possible to store states atomically. This can lead to storing duplicate states, which tends to be worsened when states are evicted, making Cuckoo hashing not practical [51]. Besides compact state storage, a second benefit of using trees with each node being stored in a single integer is that it allows arbitrarily large states to be stored atomically, i.e., a state is stored the moment the root of its tree is stored.

Because we store trees, with the individual nodes referencing each other, we do not consider alternative storage approaches, such as using a list that is repeatedly sorted, even though Alcantara *et al.* identified that using *radix-sort* [32] is competitive to hashing [1].

3 GPU programming

CUDA[2] is a programming interface that enables general purpose programming for a GPU. It has been developed and continues to be maintained by NVIDIA since 2007. In this work, we use CUDA with C++. Therefore, we use CUDA terminology when we refer to thread and memory hierarchies.

The left part of Fig. 1 gives an overview of a GPU architecture. For now, ignore the bold-faced words and the pseudo-code. A GPU consists of a finite number of *streaming multiprocessors* (SM), each containing hundreds of *cores*. For instance, a Titan RTX, which we used for this work, has 72 SMs containing together 4,608 cores. A programmer can implement functions, named *kernels*, to

[1] This refers to the *single-level* version of their Cuckoo hashing [1], which we consider in this work. Their two-level version is more complex and less efficient.

[2] https://developer.nvidia.com/cuda-zone.

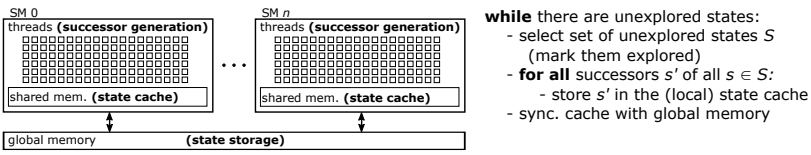


Fig. 1: State space exploration on a GPU architecture.

be executed by a predefined number of GPU threads. Parallelism is achieved by having these threads work on different parts of the data.

When a kernel is launched, threads are grouped into *blocks*, usually of a size equal to a power of two, often 512 or 1,024. Each block is executed by one SM, but an SM can interleave the execution of many blocks. When a block is executed, the threads inside are scheduled for execution in smaller groups of 32 threads called *warps*. A warp has a single program counter, i.e., the threads in a warp run in lock-step through the program. This concept is referred to as *Single Instruction Multiple Threads* (SIMT): each thread executes the same instructions, but on different data. The threads in a warp may also follow *diverging* program paths, leading to a reduction in performance. For instance, if the threads of a warp encounter an `if C then P1 else P2` construct, and for some, but not all, `C` holds, all threads will step through the instructions of both `P1` and `P2`, but each thread only executes the relevant instructions.

GPU threads can use atomic instructions to manipulate data atomically, such as a *compare-and-swap* on 32- and 64-bit integers: ATOMICCAS(`addr`, `compare`, `val`) atomically checks whether at address `addr`, the value `compare` is stored. If so, it is updated to `val`, otherwise no update is done. The actual value read at `addr` is returned.

There are various types of memory on a GPU. The *global memory* is the largest of these, 24 GB in the case of the Titan RTX, and is used to copy data between the *host* (CPU-side) and the *device* (GPU-side). It can be accessed by all GPU threads, and has a high bandwidth, but also a high latency. Having many threads executing a kernel helps to hide this latency; the cores can rapidly switch contexts to interleave the execution of multiple threads, and whenever a thread is waiting for the result of a memory access, the core uses that time to execute another thread. Another way to improve memory access times is by ensuring that the accesses of a warp are *coalesced*: if the threads in a warp try to fetch a consecutive block of memory in size not larger than the cache line (128 bytes for a Titan RTX), then the time needed to access that block is the same as the time needed to access an individual memory address.

Other types of memory are *shared memory* and *registers*. Shared memory is fast on-chip memory with a low latency, that can be used as block-local memory; the threads of a block can share data with each other via this memory. In a Titan RTX, each block can use up to 49,152 bytes of shared memory. Register memory is the fastest, and is used to store thread-local data. It is very small, though, and allocating too much memory for thread-local variables may result in data spilling over into global memory, which can dramatically limit the performance.

Finally, the threads in a warp can communicate very rapidly with each other by means of *intra-warp instructions*. There are various instructions, such as SHUFFLE to distribute register data among the threads and BALLOT to distribute the results of evaluating a predicate. Since CUDA 9.0, threads can be partitioned into *cooperative groups*. If these groups have a size that completely divides the warp size, i.e., it is a power of two smaller than or equal to 32, then the threads in a group can use intra-warp instructions among themselves.

In Section 2, we mentioned the use of buckets in a GPU hash table. When a hash table is divided into buckets, each containing $1 < n \leq 32$ elements, that still fit in the cache line, then cooperative groups of n threads each can be created, and the threads in a group can work together for the fetching and updating of buckets. This results in more coalesced memory accesses and reduces thread divergence. However, it also means that fewer tasks can be performed in parallel, and starting with the TURING architecture (2018), which the Titan RTX is built on, NVIDIA has been working on making computations less reliant on coalesced memory accessing.

4 GPU state space exploration

SLCO. For this work, we extended the state space exploration engine of GPUEXPLORE 2.0 [53] to support models of finite-state concurrent systems written in the *Simple Language of Communicating Objects* (SLCO), version 2.0 [44]. An SLCO model consists of a finite number of FSMs. The FSMs can communicate via globally shared variables, and each FSM can have its own local variables. Variables can be of type `Bool`, `Byte` and (32-bit) `Integer`, and there is support for arrays of these types. We refer with *(system) states* $s, s', \ldots$ to entire states of the system, and with *FSM states* $\sigma, \sigma', \ldots$ to the states of an individual FSM. A system state is essentially a vector, containing all the information that together defines a state of the system, i.e., the current states of the FMSs and the values of the variables.

An FSM transition $tr = \sigma \xrightarrow{st} \sigma'$ indicates that the FSM can change state from σ to σ' iff the associated *statement* st is *enabled*. A statement is either an *assignment*, an *expression* or a *composite*. Each can refer to the variables in the scope of the FSM. An assignment is always enabled, and assigns a value to a variable, an expression is a predicate that acts as a guard: it is enabled iff it evaluates to **true**. Finally, a composite is a finite sequence of statements $st_0; \ldots; st_n$, with st_0 being either an expression or an assignment, and $st_1, \ldots, st_n$ being assignments. A composite is enabled iff its first statement is enabled. A transition $tr = \sigma \xrightarrow{st} \sigma'$ can be *fired* if it is enabled, which results in the FSM atomically moving from state σ to state σ', and any assignments of st being executed in the specified order. When tr is fired while the system is in a state s, then after firing, the system is in state s', which is equal to s, apart from the fact that σ has been replaced by σ', and the effect of st has been taken into account. We call s' a *successor* of s.

The formal semantics of SLCO defines that each transition is executed atomically, i.e., cannot be interrupted by the execution of other transitions. The FSMs execute concurrently, using an interleaving semantics. Finally, the FSMs may have non-deterministic behaviour, i.e., at any point of execution, an FSM may have several enabled transitions.

State space exploration. Given an SLCO model with n FSMs, first, CUDA functions $f_1, \ldots f_n$ are generated, using a new code generator, that take as input a state s, and produce as output the successors of s which can be reached by firing a transition enabled in s of the i^{th} FSM. When the state space is generated, each state s can be analysed in parallel by n threads $t_1, \ldots, t_n$, where each t_i executes f_i to obtain some of the successors of s.

Fig. 1 presents how the different components of the state space exploration engine map on a GPU. We explain how the engine works insofar is needed. For more details, we refer the reader to [50, 51, 53]. Even though the type of input model has changed, as GPUEXPLORE only supports models without data variables, the core of the engine has remained the same.

In the global memory, a large hash table (we call it $\mathcal{G}$) is maintained to store the states visited so far. At the start, the initial state of the input model is stored in $\mathcal{G}$. Each state in $\mathcal{G}$ has a Boolean flag *new*, indicating whether the state has already been explored, i.e., whether or not its successors have been constructed.

On the right in Fig. 1, the state space exploration algorithm is explained from the perspective of a *thread block*. While the block can find unexplored states in $\mathcal{G}$, it selects some of those for exploration. In fact, every block has a *work tile* residing in its shared memory, of a fixed size, which the block tries to fill with unexplored states at the start of each exploration iteration. Such an iteration is initiated on the host side by launching the exploration kernel. States are marked as explored when added by threads to their tile.

Next, every block processes its tile. For this, each thread in the block is assigned to a particular state/FSM combination. Each thread accesses its designated state in the tile, and analyses the possibilities for its designated FSM to change state, as explained before. Hence, the threads in a group can generate successors for a single state in parallel.

The generated successors are stored in a *block-local state cache*, which is a hash table in the shared memory. This avoids repeated accessing of global memory, and local duplicate detection filters out any duplicate successors generated at the block-level. Once the tile has been processed, the threads in the block together scan the cache once more, and store the new states in $\mathcal{G}$ if they are not already present. When states require no more than 32 or 64 bits in total (including the *new* flag), they can simply be stored atomically in $\mathcal{G}$ using compare-and-swap. However, sufficiently large systems have states consisting of more than 64 bits. In this paper, we therefore focus on working with these larger states, and consider storing them as binary trees.

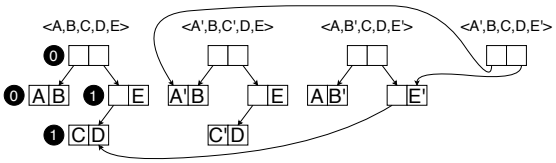


Fig. 2: An example of storing state vectors as binary trees.

5 A Compact GPU Tree Database

5.1 CPU Tree Storage

The number of data variables in a model, and their types, can have a drastic effect on the size of the states of that model. For instance, each 32-bit integer variable in a model requires 32 bits in each state. As the amount of global memory on a GPU is limited, we need to consider techniques to store states in a memory-efficient way. One technique that has proven itself for CPU-based model checkers is *tree compression* [7], in which system states are stored as binary trees. A single hash table can be used to store all tree nodes [27]. Compression is achieved by having the trees share common subtrees. Its success relies on the observation that states and their successors tend to be different in only a few data elements. In [27], it is experimentally assessed that tree compression compresses better than any other compression technique identified by the authors for explicit state space exploration. They observe that the technique works well for a multi-threaded exploration engine. Moreover, they propose an *incremental* variant that has a considerably improved runtime performance, as it reduces the number of required memory accesses to a number logarithmic in the length of the state vector.

Fig. 2 shows an example of applying tree compression to store four state vectors. The black circles should be ignored for now. Each letter represents a part of the state vector that is k bits in length. We assume that in k bits, also a pointer to a node can be stored, and that each node therefore consists of $2k$ bits. The vector <A,B,C,D,E> is stored by having a root node with a left leaf sibling <A,B>, and the right sibling being a non-leaf that has both a left leaf sibling <C,D>, and the element E. In total, storing this tree requires $8k$ bits. To store the vector <A',B,C',D,E>, we cannot reuse any of these nodes, as <A',B> and <C',D> have not been stored yet. This means that all pointers have to be updated as well, and therefore, a new root and a new non-leaf containing E are needed. Again, $8k$ bits are needed. For <A,B',C,D,E'>, we have to store a new node <A,B'> and a new root, and a new non-leaf storing E', but the latter can point to the already existing node <C,D>. Hence, only $6k$ bits are needed to store this vector. Finally, for <A',B,C,D,E'>, we only need to store a new root node, as all other nodes already exist, resulting in only needing $2k$ bits. It has been demonstrated that as more and more state vectors are stored, eventually new vectors tend to require $2k$ bits each [26, 27].

To emphasise that GPU tree compression has to be implemented vastly differently from the typical CPU approach, we first explain the latter, and the incremental approach [27]. Checking for the presence of a tree and storing it if

Algorithm 1: Tree-based Find-or-put, CPU version.

```
1 function FINDORPUT-CPU(node_t* G, node_t node):
2     if HAS-LEFT-SIBLING(node) and IS-UPDATED(LEFT-SIBLING(node)) then
3         node.left ← FINDORPUT-CPU(G, LEFT-SIBLING(node))
4     if HAS-RIGHT-SIBLING(node) and IS-UPDATED(RIGHT-SIBLING(node)) then
5         node.right ← FINDORPUT-CPU(G, RIGHT-SIBLING(node))
6     addr ← STORE(G, node)
7     return addr
```

not yet present is typically done by means of recursion (outlined by Alg. 1). For now, ignore the red underlined text. The STORE function returns the address of the given `node` in $\mathcal{G}$, if present, otherwise it stores the node and returns its address, and the FINDORPUT-CPU function first recursively checks whether the siblings of the node are stored, and if not, stores them, after which the node itself is stored. A `node` has pointers `left` and `right` to addresses of $\mathcal{G}$, and there are functions to check for the existence of, and retrieve the siblings of a node.

In the incremental approach, when creating a successor s' of a state s, the tree for s, say $T(s)$, is used as the basis for the tree $T(s')$. When $T(s')$ is created, each node inside it is first initialised to the corresponding node in $T(s)$, and the leaves are updated for the new tree. This 'updated' status propagates up: when a non-leaf has an updated sibling, its corresponding $\mathcal{G}$ pointer must be updated when $T(s')$ is stored in $\mathcal{G}$, but for any non-updated sibling, the non-leaf can keep its $\mathcal{G}$ pointer. When incorporating the red underlined text in Alg. 1, the incremental version of the function is obtained. With this version, tree storage often results in fewer calls to STORE, i.e., fewer memory accesses.

There are two main challenges when considering GPU incremental tree storage: 1) Recursion is detrimental to performance, as call stacks are stored in global memory (and with thousands of threads, a lot of memory would be needed for call stacks), and 2) The nodes of a tree tend to be spread all over the hash table, potentially leading to many random accesses. To address these, we propose a procedure in which threads in a block store sets of trees together in parallel.

5.2 GPU Tree Generation

When states are represented by trees, the tile of each thread block cannot store entire states, but it can store the roots of trees. To speed up successor generation, and avoid repeated uncoalesced global memory accessing, the trees of those roots are retrieved and stored in the shared memory (state cache) by the thread block. Once this has been done, successor generation can commence.

Fig. 3 shows an example of the state cache evolving over time as a thread generates the successor $s' =$<A,B',C,D,E'> of $s =$<A,B,C,D,E>, with the trees as in Fig. 2. Each square represents a k-bit cache entry. In addition to two entries needed to store a node, we also use one (grey) entry to store two *cache pointers* or indices, and assume that k bits suffice to store two pointers (in practice, we use $k = 32$, which is enough, given the small size of the state cache). Hence, every pair of white squares followed by a grey square constitutes one cache slot. Initially (shown at the top of the figure), the tile has a cache pointer to the root of s, of

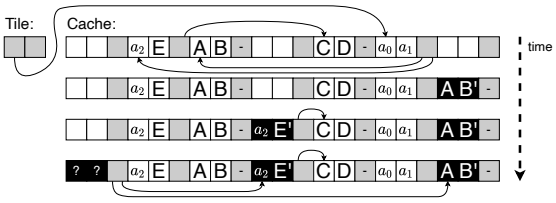


Fig. 3: Successor generation: deriving <A,B',C,D,E'> from <A,B,C,D,E>.

which we know that it contains the $\mathcal{G}$ addresses a_0 and a_1 to refer to its siblings. In turn, this root points, via its cache pointers, to the locally stored copies of its siblings. The non-leaf one contains the global address a_2. A leaf has no cache pointers, denoted by '-'. When creating s', first, the designated thread constructs the leaf <A,B'>, by executing the appropriate generated CUDA function (see Section 4), and stores it in the cache. In Fig. 3, it is coloured black, to indicate that it is marked as new. Next, the thread creates a copy of $<a_2$,E>, together with its cache pointers, and updates it to $<a_2$,E'>. Finally it creates a new root, with cache pointers pointing to the newly inserted nodes. This root still has global address gaps to be filled in (the '?' marks), since it is still unknown where the new nodes will be stored in $\mathcal{G}$.

The reason that we store global addresses in the cache is not to access the nodes they point to, but to achieve incremental tree storage: in the example, as the global address a_2 is stored in the cache, there is no need to find <C,D> in $\mathcal{G}$ when the new tree is stored; instead, we can directly construct $<a_2$,E'>. This contributes to limiting the number of required global memory accesses.

Note that there is no recursion. Given a model, the code generator determines the structure of all state trees, and based on this, code to fetch all the nodes of a tree and to construct new trees is generated. As we do not consider the dynamic creation and destruction of FSMs, all states have the same tree structure.

5.3 GPU Tree Storage at Block Level

Once a block has finished generating the successors of the states referred to by its tile, the state cache content must be synchronised with $\mathcal{G}$. Alg. 2 presents how this is done. The FINDORPUT-MANY function is executed by all threads in the block simultaneously. It consists of an outer `while`-loop (l.5-28), that is executed as long as there is work to be done. The code uses a cooperative group called `bg`, which is created to coincide with the size of a bucket (`bucketsize`). When no buckets are used, these groups can be interpreted as consisting of only a single thread each. At l.4, the `offset` of each thread is determined, i.e., its ID inside its group, ranging from 0 to the size of the group.

Every thread that still has *work to do* (l.5) enters the `for`-loop of l.7-27, in which the content of the state cache is scanned. The parallel scanning works as follows: every thread first considers the node at position $tid - \texttt{offset}$ of the cache, with *tid* being the thread's block-local ID. This node is assigned to the thread with `bg` ID 0. If that index is still within the cache limits, all threads of

Algorithm 2: Tree-based Find-or-put-many, at thread block level.

```
1  device function FINDORPUT-MANY(node_t* G):
2      node_t p, q; index_t addr; bool work_to_do ← true; bool ready; byte ballot_result
3      auto bg ← TILED-PARTITION⟨bucketsi e⟩(THIS-THREAD-BLOCK())
4      byte offset ← bg.THREAD-RANK()
5      while work_to_do do
6          work_to_do ← false
7          for i ← tid − offset; i < CACHE_SIZE; i ← i + BLOCK_SIZE do
8              ready ← false
9              if i + offset < CACHE_SIZE then
10                 p ← cache[i + offset]
11                 if IS-NEW-LEAF(p) then ready ← true
12                 else if IS-NEW-NONLEAF(p) then
13                     if LEFT-GAP(p) then
14                         cache[i + offset] ← SET-LEFT-GADDR(p, cache[LEFT-CADDR(p)])
15                     if RIGHT-GAP(p) then
16                         cache[i + offset] ← SET-RIGHT-GADDR(p, cache[RIGHT-CADDR(p)])
17                     if ¬(LEFT-OR-RIGHT-GAP(p)) then ready ← true
18                     else work_to_do ← true
19             ballot_result ← bg.BALLOT(ready)
20             while ballot_result do
21                 lane ← FIND-FIRST-SET(ballot_result) - 1; q ← bg.SHUFFLE(p, lane)
22                 addr ← FINDORPUT-SINGLE(bg, G, q)
23                 if offset = lane then
24                     ready ← false
25                     if addr = FULL then signal hash table full
26                     else SET-GADDR(cache[i], addr)
27                 ballot_result ← bg.BALLOT(ready)
28         work_to_do ← bg.BALLOT(work_to_do)
```

bg have to move along, regardless of whether they have a node to check or not. At the next iteration of the **for**-loop, the thread jumps over BLOCK_SIZE nodes as long as the index is within the cache limits.

The main goal of this loop is to check which nodes are ready for synchronisation with $\mathcal{G}$. Initially, this is the case for all nodes without global address gaps (see Subsection 5.2). Each thread first checks whether its own index is still within the cache limits (l.9). If so, the node p is retrieved from the cache at l.10. If it is a new leaf, ready is set to **true**, to indicate that the active thread is ready for storage (l.11). If the node is a new non-leaf (l.12), it is checked whether the node still has global address gaps. If it has a gap for the left sibling (l.13), this left sibling is inspected via the cache pointer to this sibling (retrieved with the function LEFT-CADDR (l.14)). The function SET-LEFT-GADDR checks whether the cache pointers of that sibling have been replaced by a global memory address, and if so, uses that address to fill the gap. The same is done for the right sibling at l.15-16. If, after these operations, the node p contains no gaps (l.17), ready is set to **true**. If the node still contains a gap, another loop iteration is required, hence work_to_do is set to **true** (l.18).

At l.19, the threads in the group perform a ballot, resulting in a bit sequence indicating for which threads ready is **true**. As long as this is the case for at least one thread, the **while**-loop at l.20-27 is executed. The function FIND-FIRST-SET identifies the least significant bit set to 1 in ballot_result (l.21), and the SHUFFLE instruction results in all threads in bg retrieving the node of the corresponding bg thread. This node is subsequently stored by bg, by calling

FINDORPUT-SINGLE (l.22) (explained later). Finally, the thread owning the node (l.23) resets its `ready` flag (l.24), and if the hash table is considered full, reports this globally (l.25). Otherwise, it records the global address of the stored node (l.26). After that, `ballot_result` is updated (l.27). Finally, once the `for`-loop is exited, the `bg` threads determine whether they still have more work to do (l.28).

5.4 Single Node Storage at Bucket Group Level

In this section, we address how individual nodes are stored by a cooperative group `bg`. Before we explain the algorithm for this, Alg. 3, in detail, we consider our options for hashing, and propose a novel combination of existing techniques.

In Section 2, we argued that Cuckoo hashing is very effective on a GPU. However, as it frequently moves elements, it is not suitable for a single hash table, since the non-leaves of a tree refer to the positions of other nodes. We address this by maintaining two hash tables, one for tree roots, and one for the other nodes, as done in [26]. The roots are then not referred to, and hence Cuckoo hashing can be applied on the root table.

In fact, when using two hash tables, we can be even more memory-efficient. In [26], it was shown that *Cleary tables* [13, 15] can be very effective to store state spaces. To handle collisions in Cleary tables, *order-preserving bidirectional linear probing* [2] is used, which involves moving nodes to preserve their order. This makes Cleary tables, like Cuckoo hashing, not suitable to store entire trees, but they can be used to store the *roots* of the trees. In a Cleary table for roots of size $2k$, each root r is hashed (bit scrambled) with a hash function h to a $2k$ bit sequence, from which $w < k$ bits are taken to be used as the *address* to store r in a table with exactly 2^w buckets, and at this position, the remaining $2k - w$ bits (the *remainder*) are actually stored. To enable decompression, h must be invertible; given a remainder and an address, h^{-1} can be applied to obtain r.

In a multi-threaded CPU context, this approach scales well [26], but the parallel approach of [26, 45] divides a Cleary table into *regions*, and sometimes, a region must be locked by a thread to safely reorder nodes. Unfortunately, the use of any form of locking, also fine-grained locking implemented with atomic operations, is detrimental for GPU performance. Further, the absence of coherent caches in GPUs means that expensive global memory accesses may be needed when a thread repeatedly checks the status of an acquired lock.

As an elegant alternative, we propose *Cleary-Cuckoo hashing*, which combines Cleary compression with Cuckoo hashing. We use m hash functions that are invertible (as with Cuckoo hashing) and capable of scrambling the bits of a root to a $2k$ bit sequence (as in Cleary tables). When we apply a function h_i ($0 \leq i < m$) on a root r, we get a $2k$ bit sequence, of which we use w bits for an address d, and store at d the remainder r' consisting of $2k - w + \lceil \log_2(m) \rceil + 1$ bits. The $\lceil \log_2(m) \rceil$ bits are needed to store the ID of the used hash function (i), and the final bit is needed to indicate that the root is *new* (unexplored). It is possible to retrieve r by applying h_i^{-1} on d and r' without the hash function ID and the *new* bit. When a collision occurs, the encountered root is evicted,

Algorithm 3: Single node find-or-put, at bucket group level.

```
1  device function index_t FINDORPUT-SINGLE(tile_t bg, node_t* G, node_t p):
2      node_t q; index_t addr
3      (q, addr) ← FOP-CUCKOO-ROOT(bg, G, p)
4      for i ← 0; q ≠ p and i < MAX_EVICT; i ← i + 1 do
5          (q, addr) ← FOP-CUCKOO-ROOT(bg, G, q)
6      return (i = MAX_EVICT? FULL; addr)

7  device function (node_t, index_t) FOP-CUCKOO-ROOT(tile_t bg, node_t* G, node_t p):
8      comprnode_t cp, cq; node_t q
9      hs ← GET-HASH-START(p); byte offset ← bg.THREAD-RANK()
10     for i ← 0; i < NUM_HASH_FUNCTIONS; i ← i + 1 do
11         (addr, cp) ← ADDR-COMPR-ROOT(p, h_(hs+i) mod NUM_HASH_FUNCTIONS)
12         (cq, pos) ← HT-FIND(bg, offset, G, addr, cp)
13         if cq = cp then return (p, addr + pos)
14         if cq = EMPTY then
15             hs ← h_(hs+i) mod NUM_HASH_FUNCTIONS
16             break
17     if i = NUM_HASH_FUNCTIONS then (cp, addr) ← ADDR-COMPR-ROOT(p, hs)
18     (cq, pos) = HT-INSERT-CUCKOO(bg, offset, G, addr, cp)
19     if cq ≠ EMPTY and cq ≠ cp then
20         q ← GET-DECOMPR-ROOT(cq, addr)
21         return (q, addr + pos)
22     return (p, addr + pos)
```

decompressed, and stored again using the hash function next in line for that root. We refer to the application of Cleary compression to roots as *root compression*.

Alg. 3 presents one version of the FINDORPUT-SINGLE function, to which a call in Alg. 2 is redirected when a root is provided. Here, $\mathcal{G}$ is a Cleary-Cuckoo table that is only used to store roots. In FINDORPUT-SINGLE, a second function FOP-CUCKOO-ROOT (l.7-22) is called repeatedly, as long as nodes are evicted or until the pre-configured `MAX_EVICT` has been reached, which prevents infinite eviction sequences (l.4). The function FOP-CUCKOO-ROOT returns the address where the given node was found or stored, and a node, which is either the node that had to be inserted or the one that was already present.

In the FOP-CUCKOO-ROOT function, lines highlighted in purple are specific for root compression, i.e., Cleary compression of roots, while the green highlighted lines concern Cuckoo hashing, addressing node eviction. The ID of the first hash function to be used for node `p`, encoded in `p` itself, is stored in `hs` (l.9), and each thread determines its `bg` offset. Next, the thread iterates over the hash functions, starting with function `hs` (l.10-16). The $\mathcal{G}$ address and node remainder are computed at l.11. If the node is new, the remainder is marked as new. If root compression is not used, we have `p` = `cp`. Then, the function HT-FIND is called to check for the presence of the remainder in the bucket starting at `addr` (l.12). If HT-FIND returns the remainder, then it was already present (l.13), and this can be returned. Note that the returned address is (`addr` + `pos`), i.e., the offset at which the remainder can be found inside the bucket is added to `addr`. Alternatively, if `EMPTY` is returned, the node is not present and the bucket is not yet full. In this case, a bucket has been found where the node can be stored. The used hash function is stored in `hs` (l.15) and the `for`-loop is exited (l.16).

At l.17, if a suitable bucket for insertion has not been found, the initial `hs` is selected again. At l.18, the function HT-INSERT-CUCKOO is called to insert `cp`.

Algorithm 4: Single node insertion, at bucket group level.

```
1  device function (comprnode_t, index_t) HT-INSERT-CUCKOO(tile_t bg, byte offset, node_t*
     G, index_t addr, comprnode_t cp):
2      comprnode_t cq ← G[addr + offset]; byte ballot_result ← bg.BALLOT(cq = cp)
3      if ballot_result then return (cp, FIND-FIRST-SET(ballot_result) - 1)
4      while ballot_result ← bg.BALLOT(cq = EMPTY) do
5          if offset = FIND-FIRST-SET(ballot_result) - 1 then
6              cq ← ATOMICCAS(G[addr + offset], EMPTY, cp)
7          cq ← bg.SHUFFLE(cq, FIND-FIRST-SET(ballot_result) - 1)
8          if cq = EMPTY or cq = cp then return (cq, FIND-FIRST-SET(ballot_result) - 1)
9          cq ← G[addr + offset]
10     byte i ← GET-EVICTION-POS(cp)
11     if offset = i then  cq ← ATOMICEXCH(G[addr + offset], cp)
12     cq ← bg.SHUFFLE(cq, i)
13     return (cq, i)
```

This function is presented in Alg. 4. Finally, if a value other than the original remainder `cp` or `EMPTY` is returned, another (remainder of a) node has been evicted, which is decompressed and returned at l.20-21. Otherwise, `p` is returned with its address (l.22). When Cuckoo hashing is not used, evictions do not occur, and at l.20-21, it is returned that the bucket is full.

Finally, we present HT-INSERT-CUCKOO in Alg. 4. The function HT-FIND is not presented, but it is almost equal to l.2-3 of Alg. 4. At l.2, each thread in `bg` reads its part of the bucket $\mathcal{G}$`[addr + offset]`, and checks if it contains `cp`, the remainder of `p`. If it is found anywhere in the bucket, the remainder with its position is returned (l.3). In the `while`-loop at l.4-9, it is attempted to insert `cp` in an empty position. In every iteration, an empty position is selected (l.5) and the corresponding thread tries to atomically insert `cp` (l.6). At l.7, the outcome is shared among the threads. If it is either `EMPTY` or the remainder itself, it can be returned (l.8). Otherwise, the bucket is read again (l.9). If insertion does not succeed, l.10 is reached, where a hash function is used by GET-EVICTION-POS to hash `cp` to a bucket position. The corresponding thread exchanges `cp` with the node stored at that position (l.11). After the evicted node has been shared with the other threads (l.12), it is returned together with its position (l.13).

6 Experiments

We implemented a code generator in PYTHON, using TEXTX [17] and JINJA2,[3] that accepts an SLCO model and produces CUDA C++ code to explore its state space. The code is compiled with CUDA 11.4 targeting compute capability 7.5. Experiments were conducted on a machine running LINUX MINT 20 with a 4-core INTEL CORE i7-7700 3.6 GHz, 32GB RAM, and a Titan RTX GPU.

The goal of the experiments is to assess how fast GPU next state computation with the tree database is w.r.t. 1) the various options we have for hashing, 2) state-of-the-art CPU tools, and 3) other GPU tools. For 2), we compare with multi-core Depth-First Search (DFS) of SPIN 6.5.1 [22] and (explicit-state) multi-core Breadth-First Search (BFS) of LTSMIN 3.0.2 [24,28].

[3] https://palletsprojects.com/p/jinja/.

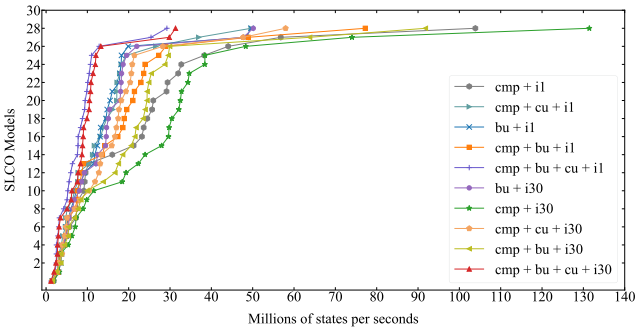


Fig. 4: Speed obtained by different GPU configurations.

In our implementation, we use 32 invertible hash functions. *Root compression* (CMP) can be turned on or off. When selected, we have a root table with 2^{32} elements, 32 bits each, and a non-root table with 2^{29} elements, 64 bits each. This enables storing 58-bit roots (two pointers to the non-root table) in $58 - 32 + \lceil \log_2(32) \rceil + 1 = 32$ bits. When using *buckets with more than one element* (CMP+BU), we have root buckets of size 8, and non-root buckets of size 16. The non-root buckets make full use of the cache line, but the root buckets do not. Making the latter larger means that too many bits for root addressing are lost for root compression to work (the remainders will be too large).

Root compression allows turning *Cuckoo hashing* on (CMP(+BU)+CU) or off (CMP(+BU)). When it is off, essentially Cleary-Cuckoo is still performed, except that evictions are not allowed, meaning that hashing fails as soon as all possible 32 buckets for a node are occupied.

In the configuration BU, neither root compression nor Cuckoo hashing is applied. We use one table with 2^{30} 64-bit elements and buckets of size 16. For reasons related to storing global addresses in the state cache, we cannot make the table larger. The 32 hash functions are used without allowing evictions.

Finally, *multiple iterations* can be run per kernel launch. Shared memory is wiped when a kernel execution terminates, but the state cache content can be reused from one iteration to the next when a kernel executes multiple iterations, by which trees already in the cache do not need to be fetched again from the tree database. We identified 30 iterations to be effective in general (i30), and experimented with a single iteration per kernel launch (i1).

With the CPU tools, we performed reachability analysis on 1- and 4-core configurations, denoted by SP-1 and SP-4 for SPIN, and LM-1 and LM-4 for LTSMIN. We only enabled state compression and basic reachability (without property checking), to favour fast exploration of large state spaces.

For benchmarks, we used models from the BEEM benchmarks [42] of concurrent systems, translated to SLCO and PROMELA (for SPIN). We scaled some of them up to have larger state spaces. Those are marked in Table 1 with '+'. Timeout is set to 3600 seconds for all benchmarks.

Table 1: Millions of states per second for various reachability tools and configurations. Pink cells: out of memory. Yellow cells: timeout. Green cell: best average. O.M.: out of memory at initialisation. SU: speedup of (CMP + i30) vs. (LM-1).

Input		CPU tools				GPUEXPLORE + SLCO Configurations								
Model	States	SP-1	SP-4	LM-1	LM-4	BITS	CR	BU + i1	CMP + i1	CMP + BU + i1	CMP + CU + i1	CMP + i30	CMP + CU + i30	SU
adding.20+	84,709,120	1.128	3.223	1.211	3.938	100	1.96	49.597	56.793	48.879	36.934	**74.026**	47.694	**61x**
adding.50+	529,767,730	0.856	O.M.	1.354	5.356	100	1.96	48.403	103.872	77.243	49.625	**131.444**	57.968	**97x**
anderson.6	18,206,917	0.623	1.362	0.516	1.309	122	1.82	14.814	16.035	13.647	11.265	**34.111**	17.649	**62x**
anderson.7	538,699,029	0.599	O.M.	0.448	1.583	141	2.75	9.309	21.192	14.244	10.426	**22.326**	10.435	**41x**
at.5	31,999,440	0.646	1.495	0.653	1.880	85	1.86	19.894	29.158	23.633	18.204	**38.457**	21.375	**59x**
at.6	160,589,600	0.454	0.869	0.695	2.387	85	1.90	17.901	38.275	27.275	19.498	**38.418**	20.359	**55x**
at.7	819,243,816	0.527	O.M.	0.666	2.372	97	1.98	12.415	**23.629**	17.381	13.194	22.329	13.378	**34x**
at.8+	3,739,953,204	0.534	O.M.	0.555	1.817	97	1.97	5.452	7.246	7.593	11.698	7.287	**11.854**	**13x**
bakery.5	7,866,401	1.400	2.570	0.410	0.904	140	2.51	11.504	7.838	7.585	6.407	**19.362**	12.782	**47x**
bakery.7	29,047,471	1.228	2.592	0.580	1.618	140	2.49	13.236	9.361	9.021	7.698	**29.783**	17.456	**51x**
bakery.8	841,696,300	0.760	1.269	0.690	2.436	140	2.40	3.745	29.410	23.957	17.116	**32.778**	18.215	**48x**
elevator2.3	7,667,712	0.554	1.099	0.463	0.985	189	3.96	4.890	3.259	3.185	2.817	**6.261**	4.827	**14x**
elevator2.4	91,226,112	0.263	0.561	0.623	1.945	213	3.97	3.025	**3.746**	2.907	3.087	3.267	2.703	**5x**
elevator2.5+	1,016,070,144	0.189	O.M.	0.473	1.630	317	5.95	1.540	**1.871**	1.545	1.520	1.839	1.491	**4x**
frogs.4	17,443,219	1.044	2.228	0.553	1.423	219	3.49	8.423	10.253	8.686	7.767	**11.549**	8.168	**21x**
frogs.5	182,772,126	0.531	1.048	0.751	2.630	251	3.84	6.766	9.573	8.214	6.898	**9.846**	6.943	**13x**
lamport.6	8,717,688	1.277	1.375	0.490	1.096	96	1.91	11.813	5.126	5.225	4.697	**27.966**	19.335	**57x**
lamport.7	38,717,846	1.001	1.822	0.672	1.979	116	1.98	18.176	23.205	18.915	16.170	**34.321**	20.641	**51x**
lamport.8	62,669,317	0.917	1.776	0.698	2.194	116	1.98	17.717	25.947	21.015	17.132	**35.387**	20.864	**50x**
loyd.2	362,880	1.278	0.758	0.255	0.497	90	1.05	**7.339**	4.204	4.220	3.723	3.243	3.930	**13x**
loyd.3	239,500,800	0.633	O.M.	0.650	2.338	114	1.96	18.268	44.073	28.970	26.556	**48.328**	28.248	**74x**
mcs.5	60,556,519	0.706	0.615	0.453	1.489	148	2.97	14.504	24.498	19.537	14.710	**29.635**	15.912	**65x**
mcs.6	332,544	1.240	0.244	0.181	0.331	156	2.75	**6.037**	3.003	3.097	2.751	3.446	3.131	**19x**
peterson.5	131,064,750	0.711	1.617	0.727	2.435	140	2.98	16.034	31.975	21.394	17.813	**32.331**	16.681	**42x**
peterson.6	174,495,861	0.852	0.756	0.720	2.451	140	2.98	15.503	32.725	22.975	17.198	**34.902**	17.030	**45x**
peterson.7	142,471,098	0.683	1.496	0.652	2.269	175	2.63	13.077	25.667	18.603	13.868	**26.183**	13.120	**37x**
phils.6	14,348,906	0.208	0.422	0.240	0.670	150	1.49	4.410	**7.458**	5.528	4.789	7.084	4.543	**30x**
phils.7	71,934,773	0.179	0.297	0.246	0.764	151	1.49	3.585	**5.702**	4.762	4.064	5.382	3.885	**22x**
phils.8	43,046,720	0.160	0.361	0.243	0.788	160	1.49	4.842	**9.151**	6.987	5.119	8.973	5.089	**37x**
szymanski.5	79,518,740	0.665	1.571	0.535	1.815	180	2.91	11.944	17.803	14.416	11.653	**18.357**	11.674	**33x**
Average		0.728	1.309	0.58	1.844	n/a		13.139	21.068	16.355	12.813	**26.621**	15.246	**40x**

Fig. 4 compares the speeds of the different GPU configurations in millions of states per second, averaged over 5 runs. For each configuration, we sorted the data to observe the overall trend. The higher the speed the better. The CMP + i30 mode (without Cuckoo hashing or larger buckets) is the fastest for the majority of models. On the other hand, it fails to complete exploration for `at.8`, the largest state space with 3.7 billion states, due to running out of memory. If Cuckoo hashing is enabled with root compression, all state spaces are successfully explored, which confirms that higher load factors can be achieved [4]. However, Cuckoo hashing negatively impacts performance, which contradicts [4]. Although it is difficult to pinpoint the cause for this, it is clear that it results from our hashing being done in addition to the exploration tasks, while in papers on GPU hash tables [1, 4], hashing is analysed in isolation. With the extra variables and operations needed for exploration, hashing should be lightweight, and Cuckoo hashing introduces handling evictions. The more complex code is compiled to a less performant program, even when evictions do not occur.

Table 1 compares GPU performance with SPIN and LTSMIN. We refer to our tool as GPUEXPLORE + SLCO. From the results of Fig. 4, we selected a set of configurations demonstrating the impact of the various options. For each model, BITS and CR gives the state vector length in bits and the compression ratio, defined as (number of roots × number of leaves per tree) / (number of nodes). With the compression ratio, we measure how effective the node sharing is, compared to if we had stored each state individually without sharing. In

Table 2: Millions of states per second for various GPU tools.

Tool	anderson.6	anderson.7	lamport.8	peterson.5	peterson.6	peterson.7	szymanski.5
GRAPPLE	2.138	14.299	n/a	10.941	9.074	8.967	n/a
GPUEXPLORE 2.0	15.863	8.737	33.063	16.874	16.705	13.581	**26.454**
GPUEXPLORE + SLCO (CMP+i30)	**34.111**	**22.326**	**35.387**	**32.331**	**34.902**	**26.183**	18.357

addition, the speed in millions of states per second is given. Regarding out of memory, we are aware that SPIN has other, slower, compression options, but we only considered the fastest, to favour the CPU speeds. Times are restricted to exploration; code generation and compilation always take a few seconds. The best GPU results are highlighted in bold. To compute the speedup (SU), the result of CMP + i30, the overall best configuration, has been divided by the LM-1 result (the single-core configuration that completely explored all state spaces except one). All GPU experiments have been done with 512 threads per block, and 3,240 blocks (45 blocks per SM). We identified this configuration as being effective for `anderson.6`, and used it for all models.

While LTSMIN tends to achieve near-linear speed-ups (compare LM-1 and LM-4), the speed of GPUEXPLORE + SLCO heavily depends on the model. For some models, as the state spaces of instances become larger, the speed increases, and for others, it decreases. The exact cause for this is hard to identify, and we plan to work on further optimisations. For instance, the branching factor, i.e., average number of successors of a state, plays a role here, as large branching factors favour parallel computation (many threads will become active quickly).

Our overall fastest configuration does not use larger buckets, nor Cuckoo hashing. Regarding buckets, as already noted in Section 3, starting with the TURING architecture, NVIDIA GPUs are less sensitive to uncoalesced accesses, and our results confirm that. Performing fewer tasks in parallel seems to be more harmful for performance than a larger number of uncoalesced accesses.

Finally, Table 2 compares GPUEXPLORE + SLCO with GPUEXPLORE 2.0 and GRAPPLE. A comparison with PARAMOC was not possible, as it targets very different types of (sequential) models. The models we selected are those available for at least two of the tools we considered. Unfortunately, GRAPPLE does not (yet) support reading PROMELA models. Instead, a number of models are encoded directly into its source code, and we were limited to checking only those models. It can be observed that in the majority of cases, our tool achieves the highest speeds, which is surprising, as the trees we use tend to lead to more global memory accesses, but it is also encouraging to further pursue this direction.

7 Conclusions and Future Work

We discussed new algorithms to achieve a GPU tree database, which enables memory-efficient explicit state space exploration for FSMs with data. We proposed Cleary-Cuckoo hashing, which makes it possible to use, for the first time, Cleary compression on GPUs. Experiments show processing speeds of up to 131 million trees per second. In the last decade, new GPUs have been increasingly effective for state space exploration [10], and in the future, they are expected to

be more capable of handling thread divergence, which still heavily occurs when accessing $\mathcal{G}$. Therefore, we are optimistic about further improvements. In the future, we will focus on optimisations and verifying temporal logic formulae.

Data Availability Statement. The datasets generated and analysed during the current study are available in the Zenodo repository [39].

References

1. Alcantara, D.A., Volkov, V., Sengupta, S., Mitzenmacher, M., Owens, J.D., Amenta, N.: Building an Efficient Hash Table on the GPU. In: GPU Computing Gems Jade Edition, pp. 39–53. Morgan Kaufmann Publishers Inc. (2012). https://doi.org/10.1016/B978-0-12-385963-1.00004-6
2. Amble, O., Knuth, D.: Ordered Hash Tables. The Computer Journal **17**(2), 135–142 (1974). https://doi.org/10.1093/comjnl/17.2.135
3. Ashkiani, S., Farach-Colton, M., Owens, J.: A Dynamic Hash Table for the GPU. In: IPDPS. pp. 419–429. ACM (2018). https://doi.org/10.1109/IPDPS.2018.00052
4. Awad, M., Ashkiani, S., Porumbescu, S., Farach-Colton, M., Owens, J.: Better GPU Hash Tables. Tech. Rep. 2108.07232, arXiV (2021). https://doi.org/10.48550/arXiv.2108.07232
5. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. JPDC **72**(9), 1083–1097 (2012). https://doi.org/10.1016/j.jpdc.2011.10.015
6. Bartocci, E., DeFrancisco, R., Smolka, S.A.: Towards a GPGPU-parallel SPIN Model Checker. In: SPIN 2014. pp. 87–96. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2632362.2632379
7. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A Database Approach to Distributed State Space Generation. Electron. Notes Theor. Comput. Sci. **198**(1), 17–32 (2008). https://doi.org/10.1016/j.entcs.2007.10.018
8. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel Probabilistic Model Checking on General Purpose Graphics Processors. STTT **13**(1), 21–35 (2011). https://doi.org/10.1007/s10009-010-0176-4
9. Bussi, L., Ciancia, V., Gadducci, F.: Towards a Spatial Model Checker on GPU. In: FORTE. LNCS, vol. 12719, pp. 188–196. Springer (2021). https://doi.org/10.1007/978-3-030-78089-0_12
10. Cassee, N., Neele, T., Wijs, A.: On the Scalability of the GPUexplore Explicit-State Model Checker. In: GaM. EPTCS, vol. 263, pp. 38–52. Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.263.4
11. Cassee, N., Wijs, A.: Analysing the Performance of GPU Hash Tables for State Space Exploration. In: GaM. pp. 1–15. EPTCS, Open Publishing Association (2017). https://doi.org/10.4204/EPTCS.263.1
12. Češka, M., Pilař, P., Paoletti, N., Brim, L., Kwiatkowska, M.: PRISM-PSY: Precise GPU-Accelerated Parameter Synthesis for Stochastic Systems. In: TACAS. LNCS, vol. 9636, pp. 367–384. Springer (2016). https://doi.org/10.1007/978-3-642-54862-8
13. Cleary, J.: Compact Hash Tables Using Bidirectional Linear Probing. IEEE Trans. on Computers **c-33**(9), 828–834 (1984). https://doi.org/10.1109/TC.1984.1676499
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009)

15. Darragh, J., Cleary, J., Witten, I.: Bonsai: A Compact Representation of Trees. Software - Practice and Experience **23**(3), 277–291 (1993). https://doi.org/10.1002/spe.4380230305
16. DeFrancisco, R., Cho, S., Ferdman, M., Smolka, S.A.: Swarm model checking on the GPU. Int. J. Softw. Tools Technol. Transf. **22**(5), 583–599 (2020). https://doi.org/10.1007/s10009-020-00576-x
17. Dejanović, I., Vaderna, R., Milosavljević, G., Vuković, Ž.: TextX: A Python tool for Domain-Specific Language implementation. Knowledge-Based Systems **115**, 1–4 (2017). https://doi.org/10.1016/j.knosys.2016.10.023
18. Edelkamp, S., Sulewski, D.: Efficient Explicit-State Model Checking on General Purpose Graphics Processors. In: SPIN. LNCS, vol. 6349, pp. 106–123. Springer (2010). https://doi.org/10.1007/978-3-642-16164-3_8
19. Edelkamp, S., Sulewski, D.: External memory breadth-first search with delayed duplicate detection on the GPU. In: MoChArt. LNCS, vol. 6572, pp. 12–31. Springer (2010). https://doi.org/10.1007/978-3-642-20674-0_2
20. García, I., Lefebvre, S., Hornus, S., Lasram, A.: Coherent Parallel Hashing. ACM Trans. Graph. **30**(6), 161 (2011). https://doi.org/10.1145/2070781.2024195
21. Holzmann, G.: The Model Checker Spin. IEEE Trans. Software Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
22. Holzmann, G., Bošnački, D.: The Design of a Multicore Extension of the SPIN Model Checker. IEEE Trans. on Software Engineering **33**(10), 659–674 (2007). https://doi.org/10.1109/TSE.2007.70724
23. Jünger, D., Kobus, R., Müller, A., Hundt, C., Xu, K., Liu, W., Schmidt, B.: WarpCore: A Library for Fast Hash Tables. In: HiPC. pp. 11–20. IEEE (2020). https://doi.org/10.1109/HiPC50609.2020.00015
24. Kant, G., Laarman, A., Meijer, J., Pol, J.v., Blom, S., Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_61
25. Khan, M., Hassan, O., Khan, S.: Accelerating SpMV Multiplication in Probabilistic Model Checkers Using GPUs. In: ICTAC. LNCS, vol. 12819, pp. 86–104. Springer (2021). https://doi.org/10.1007/978-3-030-85315-0_6
26. Laarman, A.: Optimal Compression of Combinatorial State Spaces. Innov. Syst. Softw. Eng. **15**, 235–251 (2019). https://doi.org/10.1007/s11334-019-00341-7
27. Laarman, A., van de Pol, J., Weber, M.: Parallel Recursive State Compression for Free. In: SPIN. LNCS, vol. 6823, pp. 38–56. Springer (2011). https://doi.org/10.1007/978-3-642-22306-8_4
28. Laarman, A.: Scalable Multi-Core Model Checking. Ph.D. thesis, University of Twente (2014). https://doi.org/10.3990/1.9789036536561
29. Lee, C.: Representation of Switching Circuits by Binary-Decision Programs. Bell System Technical Journal **38**, 985–999 (1959). https://doi.org/10.1002/j.1538-7305.1959.tb01585.x
30. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's Plenty of Room at the Top: What Will Drive Computer Performance After Moore's Law? Science **368**(6495) (2020). https://doi.org/10.1126/science.aam9744
31. Lessley, B.: Data-Parallel Hashing Techniques for GPU Architectures. IEEE Trans. Parallel Distributed Syst. **31**(1), 237–250 (2019). https://doi.org/10.1109/TPDS.2019.2929768

32. Merrill, D., Grimshaw, A.: High Performance and Scalable Radix Sorting: a Case Study of Implementing Dynamic Parallelism for GPU Computing. Parallel Process. Lett. **21**(2), 245–272 (2011). https://doi.org/10.1142/S0129626411000187
33. Neele, T., Wijs, A., Bošnački, D., van de Pol, J.: Partial Order Reduction for GPU Model Checking. In: ATVA. LNCS, vol. 9938, pp. 357–374. Springer (2016). https://doi.org/10.1007/978-3-319-46520-3_23
34. Osama, M.: GPU Enabled Automated Reasoning. Ph.D. thesis, Eindhoven University of Technology (2022), ISBN: 978-90-386-5445-4
35. Osama, M., Gaber, L., Hussein, A.I., Mahmoud, H.: An Efficient SAT-Based Test Generation Algorithm with GPU Accelerator. J. Electron. Test. **34**(5), 511–527 (2018). https://doi.org/10.1007/s10836-018-5747-4
36. Osama, M., Wijs, A.: Parallel SAT Simplification on GPU Architectures. In: TACAS. LNCS, vol. 11427, pp. 21–40. Springer (2019). https://doi.org/10.1007/978-3-030-17462-0_2
37. Osama, M., Wijs, A.: SIGmA: GPU Accelerated Simplification of SAT Formulas. In: IFM. LNCS, vol. 11918, pp. 514–522. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_29
38. Osama, M., Wijs, A.: GPU Acceleration of Bounded Model Checking with ParaFROST. In: CAV, Part II. LNCS, vol. 12760, pp. 447–460. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_21
39. Osama, M., Wijs, A.: Artifact for A GPU Tree Database for Many-Core Explicit State Space Exploration (2023). https://doi.org/10.5281/zenodo.7509129
40. Osama, M., Wijs, A., Biere, A.: SAT Solving with GPU Accelerated Inprocessing. In: TACAS. LNCS, vol. 12651, pp. 133–151. Springer (2021). https://doi.org/10.1007/978-3-030-72016-2_8
41. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: ESA. LNCS, vol. 2161, pp. 121–133. Springer (2001). https://doi.org/10.1007/3-540-44676-1_10
42. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN 2007. LNCS, vol. 4595, pp. 263–267 (2007). https://doi.org/10.1007/978-3-540-73370-6_17
43. Prevot, N., Soos, M., Meel, K.: Leveraging GPUs for Effective Clause Sharing in Parallel SAT Solving. In: SAT. LNCS, vol. 12831, pp. 471–487. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_32
44. de Putter, S., Wijs, A., Zhang, D.: The SLCO Framework for Verified, Model-driven Construction of Component Software. In: FACS. Lecture Notes in Computer Science, vol. 11222, pp. 288–296. Springer (2018). https://doi.org/10.1007/978-3-030-02146-7_15
45. van der Vegt, S., Laarman, A.: A Parallel Compact Hash Table. In: MEMICS. LNCS, vol. 7119, pp. 191–204. Springer (2011). https://doi.org/10.1007/978-3-642-25929-6_18
46. Wei, H., Chen, X., Ye, X., Fu, N., Huang, Y., Shi, J.: Parallel Model Checking on Pushdown Systems. In: ISPA/IUCC/BDCloud/SocialCom/SustainCom. pp. 88–95. IEEE (2018). https://doi.org/10.1109/BDCloud.2018.00026
47. Wei, H., Ye, X., Shi, J., Huang, Y.: ParaMoC: A Parallel Model Checker for Pushdown Systems. In: ICA3PP. LNCS, vol. 11945, pp. 305–312. Springer (2019). https://doi.org/10.1007/978-3-030-38961-1_26
48. Wijs, A., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: SPIN. LNCS, vol. 7385, pp. 98–116. Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_9

49. Wijs, A.: BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In: CAV, Part II. LNCS, vol. 9780, pp. 472–493. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_26
50. Wijs, A., Bošnački, D.: GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In: TACAS. LNCS, vol. 8413, pp. 233–247 (2014). https://doi.org/10.1007/978-3-642-54862-8_16
51. Wijs, A., Bošnački, D.: Many-Core On-The-Fly Model Checking of Safety Properties Using GPUs. STTT **18**(2), 169–185 (2016). https://doi.org/10.1007/s10009-015-0379-9
52. Wijs, A., Katoen, J.P., Bošnački, D.: Efficient GPU Algorithms for Parallel Decomposition of Graphs into Strongly Connected and Maximal End Components. Formal Methods Syst. Des. **48**(3), 274–300 (2016). https://doi.org/10.1007/s10703-016-0246-7
53. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU Explicit-State Model Checking. In: FM. LNCS, vol. 9995, pp. 694–701. Springer (2016). https://doi.org/10.1007/978-3-319-48989-6_42
54. Wu, Z., Liu, Y., Liang, Y., Sun, J.: GPU Accelerated Counterexample Generation in LTL Model Checking. In: ICFEM. LNCS, vol. 8829, pp. 413–429. Springer (2014). https://doi.org/10.1007/978-3-319-11737-9_27
55. Wu, Z., Liu, Y., Sun, J., Shi, J., Qin, S.: GPU Accelerated On-the-Fly Reachability Checking. In: ICECCS. pp. 100–109 (2015). https://doi.org/10.1109/ICECCS.2015.21
56. Youness, H., Osama, M., Hussein, A., Moness, M., Hassan, A.M.: An Effective SAT Solver Utilizing ACO Based on Heterogenous Systems. IEEE Access **8**, 102920–102934 (2020). https://doi.org/10.1109/ACCESS.2020.2999382
57. Youness, H.A., Ibraheim, A., Moness, M., Osama, M.: An Efficient Implementation of Ant Colony Optimization on GPU for the Satisfiability Problem. In: PDP. pp. 230–235. IEEE (2015). https://doi.org/10.1109/PDP.2015.59

Author Index

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 705–708, 2023.
https://doi.org/10.1007/978-3-031-30823-9