Finite Automata Methods for Automated Reasoning

Habilitation Thesis

Lukáš Holík

Brno, Autumn 2020

Abstract

This habilitation thesis summarises the results of the author around algorithmic techniques for finite automata over finite words and trees and their applications in automated reasoning. Finite automaton is a core concept of computer science that comes with a versatile toolbox facilitating simple and elegant solutions to many problems, with strong theoretical guarantees in the sense of complexity and completeness. Automata research has been constantly delivering results relevant to a range of domains such as automated reasoning, pattern matching, formal verification, language processing, databases, web technologies, etc. The practical impact of research in automata has been however lagging behind the theoretical one, often due to an insufficient scalability of automata algorithms. This thesis summarises our work on improving scalability and general usability of automata technology. It is focused on efficient heuristics for general automata problems as well as on utilising application opportunities for automata and application specific techniques, mainly in verification of pointer programs, parallel programs, string manipulating programs, deciding logics WS1S and WSkS, and pattern matching. Some results on related models of concurrent systems such as Petri nets and well-structured systems are also mentioned.

Keywords

Finite automata, tree automata, non-determinism, alternation, size reduction, counting automata, concurrency, Petri Nets, well-structured systems, pointer programs, parallel pointer programs, formal verification, decision procedures, shape analysis, tree automata, separation logic, string constraints, monadic second-order logic, WS1S, WSkS, pattern matching.

Acknowledgment

I thank my Ph.D. advisor Tomáš Vojnar and my postdoc mentor Parosh Aziz Abdulla for guiding me through the beginnings of this work and my academic career. The results discussed here were achieved together with my colleagues and students, who also made the process an overwhelmingly positive experience. My thanks to Mohamed Faouzi Atig, Ahmed Bouajjani, Milan Češka (both), Yu-Fang Chen, Lorenzo Clemente, Tomáš Fiedor, Peter Habermehl, Vojtěch Havlena, Frédéric Haziza, Martin Hruška, Petr Janků, Bengt Jonsson, Lisa Kaati, Filip Konečný, Ondřej Lengál, Anthony Widjaja Lin, Roland Meyer, Richard Mayr Bui Phi Diep, Cong Qui Trinh, Ahmed Rezine, Othmanne Rezine, Adam Rogalewicz, Philipp Rümmer, Jiří Šimáček, Marek Trtík, Lenka Turoňová, Margus Veanes, Sebastian Wolff.

Over the time, I was supported by a number of grant projects, mainly by the Czech Science Foundation and Ministry of Education, Youth and Sports of Czech Republic.

Contents

Chapter 1

Introduction

This habilitation thesis is a brief overview of the authors research. It is centered around results on algorithmic techniques for finite word and tree automata, on their applications in automated reasoning, mainly program verification and deciding related logics, and it also touches on extensions of automata such as Petri nets, well-structured systems, grammars, or automata on infinite words. Finite automaton is a basic mathematical model of a computing machine. Automata research has been influential (for instance, in that 6 Turing award winners, Rabin, Scott, Pnueli, Sifakis, Clarke, and Emerson, worked in automata-related fields in the last 30 years), tens or hundreds of extensions of the basic automata model were proposed and automata were used to derive a plethora of strong theoretical results applicable in many areas. Besides the most prominent applications of basic variants of automata in pattern matching and compiler construction, the application possibilities include deciding rich logics about finite words, trees, sets, graphs, or integer arithmetic [77, 93, 95, 116, 118, 134, 159, 238, 254, 255]; model checking of finite state systems [63, 95, 134] as well as infinite state systems [57, 59, 84, 155, 187, 213]; synthesis [156]; safety and security analysis of string and pointer programs [28, 172, 194, 201, 206]; type-checking general XML transformations [60, 91, 212, 251, 253]; reasoning about UML schemes [62, 160, 161, 188]; compressing XML files, program code, or digital images [120, 250], and others. The reach of extended variants of automata is even greater: Due to the famous results like those of Büchi and Rabin, automata over infinite words and trees can be used to decide extremely expressive fragments of monadic second order logic (MSO) [94, 95, 134, 226], to test program behavior wrt complex patterns expressible in temporal logics such as linear temporal logic (LTL) [111, 112, 137, 220, 225, 267], and even a fairly general automated synthesis of programs [221, 227]. Numerous automata extensions support reasoning about properties of computations such as time, costs, or probabilities over various kinds of infinite alphabets like integers, reals, strings.

Only a fraction of this potential is however utilised in practice. The major obstacle is that the central computational problems in automata theory are of a high worst-case complexity (typically ranging from PSPACE-complete to non-elementary) and the textbook algorithms do not scale well enough in practical instances. Around 2006, when the work discussed here was starting, most of successful automata technology was based on textbook algorithms for deterministic automata [77, 88, 159], and it was reaching its limits. This was most visible in the more ambitious application areas such as deciding general logics, verification, or synthesis.

A good example is the tool MONA [133, 185], the best-known solver for the logics WS1S and WS2S, that implements most of the standard algorithms for deterministic automata. MONA has found numerous applications in verification of programs with dynamic linked data structures [109, 200, 201, 213, 284], strings [252], arrays [287], of parametric systems [69, 75, 89], distributed systems [184, 244], in hardware verification [68], automated synthesis [156, 169, 235], or computational linguistics [214]. MONA adds to textbook algorithms a rich set of heuristics and an outstanding engineering, which helped it to stay unbeaten for about 20 years. Despite that, the high worst-case complexity often strikes back, forcing the users to seek workarounds, restrict their input, or abandon the automata approach altogether (e.g. [200, 201, 273]).

The impulse that initiated the research discussed in this thesis was the appearance of the antichain algorithms for non-deterministic automata [277]. They could solve language inclusion and universality dramatically faster then the most optimised textbook algorithms for deterministic automata, showing that non-determinism combined with smart heuristics can lead to substantial efficiency gains. We started by elaborating on the antichain idea and later branched the research by included concepts such as simulation reduction, abstraction, variants of symbolic representation, non-determinism or alternation. In some cases, we indeed succeeded in significantly extending scalability of automata methods, and in using them to advance state of the art in practical application domains such as pointer program verification, string program analysis, or pattern matching. The thesis will briefly outline these results.

Plan of the thesis. Chapter 2 summarises our work on efficient solutions for classical finite automata problems. We explain our initial motivations in regular model checking in Section 2.1, then we present results on simulation-based size-reduction of automata in Section 2.2, antichain algorithms in Section 2.3, use of alternation in Section 2.4, and on counting in Section 2.5. Chapter 3 outlines our work in applications of automata in deciding the logics WS1S and WSkS (Section 3.1), in verification of pointer programs (Section 3.2), in string solving (Section 3.3), and in pattern matching (Section 3.4). Chapter 4 outlines work on analysis of concurrent systems. Each section is concluded by the list of the author's publications contributing to the topic. Several representative papers are attached in Appendix A.

Chapter 2

Automata Algorithms

In this chapter, we will discuss our results on algorithms and techniques for classical automata problems. We will first explain the initial motivation of this work in abstract regular model checking and then summarise the main results on simulation-based reduction, antichain algorithms, and means of succinct automata representation such as alternation or counters.

2.1 At the Beginning Was Regular Model Checking

The initial phase of this work was motivated by the method of *abstract regular model checking* (*ARMC*), which is a versatile automata-based verification framework applicable in a wide range of systems. ARMC belongs to the family of techniques of regular model checking, first mentioned probably in [181], that approximate state spaces or the transition relations of a system as a regular set (or as rational relations, respectively). The idea was elaborated on in a large number of flavours, e.g. [55, 56, 76, 88, 108, 123, 177, 257] to name few, and continues to find applications and being rediscovered (lately for in stance in the context of analysis of string constraints [18, 107, 280]).

ARMC [87] particularly computes the set of all configurations reachable in a system with the set of initial configuration I given as an automaton (word or tree) and the transition relation δ given in the form of a transducer or as a special procedure that manipulates automata structure. The task is then to compute the automaton representing the set $\delta^*(I)$ of all reachable configuration in the form of the fixpoint $\mu X.\, I \cup \delta(X)$. ARMC uses an overapproximating abstraction over the automata structure to accelerate the computation and achieve convergence, and a counterexample guided refinement to adjust the abstraction. The abstraction refinement uses backward concrete run from bad states in order to find whether, where, and how did the abstraction caused their occurrence. ARTMC [84] is a generalisation of ARMC to tree automata that allows reasoning about systems with complex graph configurations such as pointer programs or communication protocols with non-trivial topologies,

or XML documents. AR(T)MC uses heavily automata operations such as product construction, determinization, minimization, and language inclusion test. It is, as most of similar automata techniques at the time, originally based on deterministic automata. Determinization and forms of product construction are called frequently and so state explosion is a major bottleneck, despite that deterministic minimization is used to keep automata size at bay.

The works of De'Wulf and Raskin [277] on antichain algorithms for inclusion testing of non-deterministic finite word automata (NFA) showed that the potentially much smaller NFA can be used efficiently as symbolic representations of sets. Inspired by these works, we started to search for possibilities of building the entire framework of ARTMC on non-deterministic tree automata. The main ingredients that allowed us to achieve the needed efficiency were generalisation of the antichain algorithms and also of simulation-based reduction to tree automata [2, 14]. This indeed brought a huge scalability improvement compared to the earlier deterministic version, confirming that non-determinism may really work in practice when treated with care. The implementation of the ARTMC framework of [14], even though rather basic, is still among the most efficient implementation of regular model checking (as also recently observed in [108]).

2.2 Simulation-based Reduction

Minimization or size reduction[1] is useful in almost all applications of finite automata. It is especially important in applications such as ARMC or in deciding automata related logics like WSkS, where automata are created by sequences of heavy automata construction such as product or subset construction. Since the increase of automata is after each single operation carried over to the subsequent steps, even a moderate increase per step may mean an overall super exponential explosion. Using minimization or size reduction after every step is then indispensable.

The size of non-deterministic automata can be decreased by merging states (replacing two states by one that inherits all incoming and outgoing transitions of the original states). States with the same forward language can be merged (words accepted by a run starting at a state) as well as states with the same backward language (words accepted by a run ending at a state) while preserving the language. Computing language equivalence/inclusion between states is however expensive, PSPACE-complete for word automata and EXPTIME-complete for tree automata. A cheaper feasible alternative is the well known technique of simulations and bisimulations-based automata reduction (e.g. [97, 115, 162, 171]). Simulation relations can be computed in

[1] To *minimize* is usually understood as a to compute the unique minimal automaton, which exists for deterministic automata. In the context of non-deterministic automata, where a unique minimal automaton needs not exist, we speak about *size reduction*.

the time mn for finite word automata, where m is the number of transitions and n is the number of states. Bisimulation can be computed in time $m \log n$ [217, 266]. Modern algorithms such as [102, 228] can cut the complexity of computing simulation down even more (the number of states n is replaced by the number of simulation equivalence classes) and are very fast in practice. Algorithms for computing bisimulation use similar techniques and are even faster. Simulation is a coarser relation then bisimulation and allows to reduce automata more. In our experience, the additional cost of simulation is well worth the better reduction, and so we focused on simulations.

The basic definition of the word automata *simulation preorder* $\preceq$ (the maximal simulation, we call it simply simulation here) is the following: it is the maximal relation satisfying that (1) if $q \preceq r$, then for all a-successors q' of q, there is an a-successor r' of r' that simulates q, and that (2) a non-final state cannot simulate a final state. In other words, it is the largest relation where non-final states cannot simulate final ones and which is monotonic with respect to the transition relation (bisimulation is then a simulation which is symmetrical). In the definition of *backward simulation*, the transition relation is replaced by its reverse and final states are replaced by initial ones. The standard version is then called *forward simulation*.

Our first steps toward non-deterministic automata-based ARTMC was to generalize simulation-based reduction to tree automata (bisimulation relation already existed [54, 164], but it did not provide satisfactory reduction). Simulations and bisimulation in tree automata do not have symmetrical upward and downward variants, corresponding to forward and backward for words—trees do not look the same from the top as from the bottom. The easier variant is the *downward simulation*, which underapproximates the inclusion of downward languages of sates (languages of trees accepted from the state at their root by following transitions downwards). Namely, state q simulates r only if for every tree transition $q \xrightarrow{a} (q_1, \ldots, q_n)$ of q, there is a transition $r \xrightarrow{a} (r_1, \ldots, r_n)$ where each state q_i is simulated by the state r_i.[2] The *upward simulation*, the tree counterpart of the word backward simulation, is more complex and technical (see [2] for details).

Both tree automata simulations can be used for size reduction by merging simulation equivalent states. We have also discovered that besides the standard merging of forward/backward or upward/downward-simulation equivalent states, the two types of relations can be combined, giving a rise to a new relation, which we call *mediated preorder*, that can be coarser and can still be used to merge equivalent states while preserving automata language. The idea can be roughly explained as follows. Standard merging of simulation equivalent states can be seen allowing the automaton jump from a state to any simulation equivalent state (it could be simulated by introducing ϵ-transitions between simulation equivalent states). This does not alter the

[2] We will be writing tree automata rules in the top-down manner.

language since for instance in the case of forward word simulation, the simulation equivalent state at the end of the jump can further accept the same word as the source state of the jump and vice versa. Merging of states equivalent in the mediated preorder pushes the idea a little further. Roughly, the automaton can be allowed to jump from a state q to r if there is a witness that prefixes accepted in q connected with suffixes accepted from r are already in the language (hence jumping from q to r cannot lead to accepting a word outside the language). Such witness may be a state s which backward simulates q and forward simulates r. Forward and backward word simulation and upward and downward tree simulation can be combined into the mediated equivalence where witnesses are guaranteed to exist for equivalent states and the states can be merged without affecting the language. Mayr and Clemente later elaborated on these ideas even further in [113–115].

A somewhat unexpected by-product of our work on simulation minimization was a technique of computing tree automata simulations by means of reduction to computation of the standard word automata simulations. Advanced word automata algorithms as [102, 228] can be then used to compute tree automata simulations, for no extra cost. The translation for the downward simulation particularly works as follows. A tree automaton is translated into a word automaton that has transitions between the original states and the former left hand sides of transitions: For an original tree transition $(q_1, \ldots, q_n) \xrightarrow{a} q$, all states q_i will originate an "unnamed" transition to the state $(q_1, \ldots, q_n)$, which in turn will originate an a-transition to q. The simulation preorder on the word automaton, restricted to the original states, is then the simulation preorder on the tree automaton.

We have achieved similar results with tree automata bisimulations, which are coarser but easier to compute, and studied ways of combining simulations with bisimulations to achieve trade-offs between speed and reduction [4, 7]. Later studied some more advanced forms of simulations in context of tree automata in [8] (such as lookahead simulation).

Last, we transferred these results to alternating Büchi word automata in [3]. Alternating automata are similar to tree automata in that their runs are also trees. Their transitions are similar to tree automata transitions. The difference from tree automata that matters in the definitions of simulations is only that the left-hand side tuples of states in transitions of alternating automata are not ordered. We have defined upward and downward simulations and their combinations analogously as for tree automata and shown that they can be used for merging states. We also found means of computing alternating automata simulations by means of computing normal automata simulations, similarly as in [2] for tree automata. Although the main results in [3] are a quite close analogy to [2], the considered Büchi acceptance condition together with alternation made the correctness proof in [3] quite challenging.

Contributed Papers[3]

[1] Parosh A. Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. "Composed Bisimulation for Tree Automata". In: *Proc. of CIAA'08*. Vol. 5148. LNCS. Springer Berlin Heidelberg, 2008, pp. 212–222.

[2] Parosh A. Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. "Computing Simulations over Tree Automata". In: *Proc. of TACAS'08*. Vol. 4963. LNCS. Springer Berlin Heidelberg, 2008, pp. 93–108.

[3] Parosh A. Abdulla, Yu-Fang Chen, Lukáš Holík, and Tomáš Vojnar. "Mediating for Reduction (on Minimizing Alternating Büchi Automata)". In: *Proc. of FSTTCS'09*. Vol. 4. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2009, 1–12.

[4] Parosh Aziz Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. "Composed Bisimulation for Tree Automata". In: *International Journal of Foundations of Computer Science* 04 (2009), pp. 685–700.

[5] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. "Simulation Subsumption in Ramsey-Based Büchi Automata Universality and Inclusion Testing". In: *Proc. of CAV'10*. Vol. 6174. LNCS. Springer Berlin Heidelberg, 2010, pp. 132–147.

[6] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, and Tomáš Vojnar. "Mediating for reduction (on minimizing alternating Büchi automata)". In: *Theoretical Computer Science* 552.0 (Oct. 2014), pp. 26–43.

[7] Parosh Aziz Abdulla, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. "A Uniform (Bi-) Simulation-Based Framework for Reducing Tree Automata". In: *Electron. Notes theor. Comput. Sci.* 251 (2009), pp. 27–48.

[8] **Ricardo Almeida, Lukáš Holík, and Richard Mayr. "Reduction of Nondeterministic Tree Automata". In: *Proc. of TACAS'16*. Vol. 9636. LNCS. Springer, 2016, 717–735.**

[9] Lukáš Holík. "Simulations and Antichains for Efficient Handling of Finite Automata". PhD thesis. Brno, CZ: Department of Intelligent Systems FIT BUT, 2011.

[10] Lukáš Holík and Jiří Šimáček. "Optimizing an LTS-Simulation Algorithm". In: *Proc. of MEMICS'09*. Znojmo, CZ: Faculty of Informatics MU, 2009, 93–101.

[11] Lukáš Holík, Ondřej Lengál, Juraj Síč, Margus Veanes, and Tomáš Vojnar. "Simulation Algorithms for Symbolic Automata". In: *Proceedings of ATVA'18*. Vol. 11138. LNCS. Springer, 2018, pp. 109–125.

2.3 Antichain Algorithms

The antichain algorithms was the other research direction that we persuaded in the context of ARTMC. They are namely useful in language inclusion checking used to detect convergence of the fixpoint computation. The antichain principle was discovered by Doyen, De Wulf, and Raskin in their papers [276,

[3] The work [8] (in bold) is attached to this thesis.

277] where it was applied to solving games of imperfect information and to testing finite word automata language inclusion and universality. The idea is best explained on testing finite word automata language universality (is the language Σ^*?). The classical algorithm builds the deterministic automaton accepting the complement language by the subset construction and then tests that it has no reachable accepting state (in other words, is a set of states not containing a final state reachable in the automaton created by the subset construction?). The number of subsets is however at most exponential and the construction may explode. The first observation leading to the antichain algorithm is that in order to test universality, it is not necessary to build the entire deterministic machine. Its state space can instead be searched on-the-fly, while it is being built, and parts that are clearly irrelevant to whether the target will be reached or not can be omitted. Namely, in the deterministic automaton accepting the complement language, larger subset states *simulate* the smaller ones, and so if an accepting state (not containing final states) is reachable from the larger state, then it is reachable from the smaller state too. We say that the smaller set *subsumes* the larger state, and the subsumed larger state can be discarded. The algorithm thus builds a much reduced state space, only the antichain of the smallest states. This dramatically decreases the running times.[4]

To enable non-deterministic automata-based ARTMC, we have generalised antichain-based universality and inclusion testing from [277] to tree automata [14]. We first focused on extending well known bottom-up tree automata algorithms and subset construction (see [116]) resulting in an algorithm analogous to the original word automata version. We have then shown in [13] that antichain algorithms (for word as well as tree automata) can utilise precomputed simulation relations for even more aggressive subsumption pruning. Namely, in the simplest case of universality testing, a set subsumes another if for every state in the subsuming set, there is a state in the subsumed set that simulates it. Depending on the size of the simulation relation, this definition of subsumption may be much more liberal then the original relation of being a subset. This basic idea becomes a little more involved when it comes to inclusion testing.[5] We have then elaborated also on the top-down algorithm for inclusion testing of tree automata combined with antichains and simulation in [3, 15]. The basic principle of the top-down inclusion testing is much less obvious then the bottom-up approach. It turned out that we actu-

[4] We note that, in the spirit, the antichain algorithms can be but into the category of property driven reachability algorithms, which is a term nowadays used for modern model checking algorithms PDR/IC3 [90, 163]. The antichain idea can also be seen as a specialisation of the techniques used in deciding coverability in Petri nets and generally well-structured systems [53] (Indeed, a relation is a simulation if and only if the transition relation is monotone with respect to it. The subset construction produces a monotone well-structured system.).

[5] Raskin and Doyen arrived independently to almost the same result and published it simultaneously at the same conference [131].

ally rediscovered it, it was published before in [167], in the context of analysis of XML manipulations. The combination with the antichain principle and other technical solutions in [15] were however still new.

Based on the work [140], that uses antichains to speed-up the Ramsey-based algorithm to Büchi universality testing [96, 243], we have noticed that simulations can be combined with antichains also in the world of infinite words. The Ramsey-based algorithms build a set of so called transition profiles. A transition profile is the set of pairs of states which characterises the automaton's behaviour on some word: the word takes the automaton from any right state of a pair in the relation to its left state (it is an element of the so called transition monoid of the automaton). The antichain principle can be used to prune the space of the transition profiles [140]. We found that the pruning can also be strengthened based on simulation, and that even combining variants of forward and backward simulation is possible, justified by similar principles as those discussed in Section 2.2 in context of combining forward and backward simulation for reducing automata size [5, 209]. Since the Büchi automata algorithms are more expensive then their word automata counterparts, and tend to explode more, using the simulation relations pays off more than in the case of word automata.

This line of work was later continued in works on testing of inclusion of context free grammars [16, 17] in a regular language and later solving context free games with regular objectives. Here the transition profiles are used to summarise non-terminals of the grammar (similar to procedure summaries) and the antichain principle can be used analogously as in [5, 140, 209].

Contributed Papers[6]

[5] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. "Simulation Subsumption in Ramsey-Based Büchi Automata Universality and Inclusion Testing". In: *Proc. of CAV'10*. Vol. 6174. LNCS. Springer Berlin Heidelberg, 2010, pp. 132–147.

[12] Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. "Advanced Ramsey-Based Büchi Automata Inclusion Testing". In: *Proc. of CONCUR'11*. Vol. 6901. LNCS. Springer Berlin Heidelberg, 2011, pp. 187–202.

[13] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. "When Simulation Meets Antichains". In: *Proc. of TACAS'10*. Vol. 6015. LNCS. Springer Berlin Heidelberg, 2010, pp. 158–174.

[14] Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar. "Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata". In: *Proc. of CIAA'08*. Vol. 5148. LNCS. Springer Berlin Heidelberg, 2008, pp. 57–67.

[6] The work [15] (in bold) is attached to this thesis.

[15] **Lukáš Holík, Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. "Efficient Inclusion Checking on Explicit and Semi-symbolic Tree Automata". In: *Proc. of ATVA'11*. Vol. 6996. LNCS. Springer Berlin Heidelberg, 2011, pp. 243–258.**

[16] Lukáš Holík and Roland Meyer. "Antichains for the Verification of Recursive Programs". In: *Proc. of NETYS'15*. Vol. 9466. LNCS. Springer, 2015, 322–336.

[17] Lukáš Holík, Roland Meyer, and Sebastian Muskalla. "Summaries for Context-Free Games". In: *Proc. of FSTTCS'16*. Vol. 65. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 41:1–41:16.

2.4 Alternation

Alternating automata (originating in [92, 104]) extend the expressive means of non-deterministic automata by allowing conjunctive transitions. Non-determinism allows disjunction, that is "the word must be accepted by following either this or that transition", and alternation allows to say "the word must be accepted by following this as well as that transition". Alternating automata allow succinct representation of intersection (conjunction) and also complementation (negation). Complementation can be done by complementing the acceptance of states and swapping the polarity of logical operators in the transition relation. Boolean automata [92] are then an even more succinct variant which also allows to negate transitions. The price for succinctness of alternating automata is paid in testing emptiness which becomes exponential (PSPACE-complete for finite word alternating automata). A possible way of testing language emptiness is to first non-determinize the alternating automaton and then test its emptiness through the linear search for an accepting state. The non-determinization can explode exponentially, similarly as subset construction in determinization. Therefore, we have been trying to alleviate the cost of emptiness test by using smart heuristics in testing emptiness directly on the alternating automata, without the prior non-determinization. Antichain algorithms are applicable, since non-determinization is a variation on the subset construction. [145] has proposed an interesting possibility of combining antichains with abstraction refinement. Also the recent congruence algorithms [78] were tried with alternating automata in [121].

Us [18] and also others [119, 144, 270] were experimenting with using modern model checking algorithms. We were namely focusing on the relatively new IC3/PDR algorithm [163] applied to alternating automata emptiness in the context of string solving (we will discuss string solving more in Section 3.3). The translation from an alternating automata emptiness into the model checking problem is fairly straightforward and the results show a great potential of similar techniques. A meaningful comparison of existing algorithms is needed, and it is a part of our immediate future work.

Contributed Papers[7]

[3] Parosh A. Abdulla, Yu-Fang Chen, Lukáš Holík, and Tomáš Vojnar. "Mediating for Reduction (on Minimizing Alternating Büchi Automata)". In: *Proc. of FSTTCS'09*. Vol. 4. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2009, 1–12.

[18] **Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. "String constraints with concatenation and transducers solved efficiently". In: *PACMPL* 2.POPL (2018), 4:1–4:32.**

2.5 Counting

Recently we have started investigating the possibility of using counting in automata for succinctness in the context of regular pattern matching. Regular expressions often use bounded repetition where e.g. the regex `(ab){100}` means 100 repetitions of `ab`. Similar bounded repetitions arise other applications of automata too: WS1S formulae may talk about repetitions of certain patterns as well as automata encodings of Presburger formulae [275], string constraints generated from string programs talk about abut numeric constraints on lengths of strings, automata from the program verification framework of [158] may describe a control flow with a bounded for-loop as a bounded repetitions of the automaton describing the body. These properties are still regular, but the size of the automaton is linear to the repetition count, hence exponential the numeral that specifies it. Using standard automata methods then becomes problematic.

We have therefore started investigating variants of simple counter automata, that we call counting automata, that combine counting with finite state control to express such structural repetitions concisely. They are modest variants of counter automata characterised by a very limited set of allowed operations on the counters (essentially only increment, reset to 0, and comparison with a constant) and a bounded state space. Similar automata were considered in works on pattern matching before for instance in [168, 183, 245, 248]. Our first results concern succinct determinization of these automata in order to facilitate fast regular pattern matching [19, 20] (discussed in Section 3.4). One of our goals is to generalise all basic automata algorithms (such as boolean operations and minimization/reduction) in a way that would keep the succinctness of counters in order to make this kind of automata applicable in a range of domains.

[7] The work [18] (in bold) is attached to this thesis.

Contributed Papers[8]

[19] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. "Succinct Determinisation of Counting Automata via Sphere Construction". In: *Proc. of APLAS'19*. Vol. 11893. LNCS. Springer, 2019, pp. 468–489.

[20] **Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. *Regex Matching with Counting-Set Automata*. accepted at OOPSLA'20. 2020.**

2.6 Future Directions

One of the most obvious future directions is to implement a well engineered library that would include efficient techniques discussed above, be efficient and easily modifiable by others. It would also provide a platform for much needed comparison of newly emerging algorithms, for instance those for language inclusion or alternating automata language emptiness checking (there are the antichain algorithms [15, 131, 277], the PDR/IC3-based approaches [90, 163], an algorithm that uses abstraction refinement automata structure [145], and the modulo congruence algorithms [78, 121], which were proposed in different contexts and never compared). We believe the ideas on the property driven search as well the idea of abstracting the automata structure are very promising. Another interesting direction is combining approaches as antichains and property driven reachability with deterministic minimization or simulation-based size reduction (minimization is of of the most crucial optimization of the frameworks such as those of MONA and AR(T)MC, but it cannot be combined with the aforementioned techniques out of the box). We also wish to elaborate on the idea of succinct representation of counting constraints from [19, 20] and develop a complete toolbox for counting or counting set automata. More inspiration for designing further automata heuristics may be drawn from the successful algorithms used in SAT solving. Computation of truly minimal non-deterministic automata, which could go beyond reduction by merging and similar techniques, is also a project which could yield interesting theoretical and perhaps also practical results. Generally, we wish to improve efficiency of automata in practice and so we will be working in the context of practical applications, continuing with the application domains discussed in Chapter 3 and search for new ones.

[8] The work [20] (in bold) is attached to this thesis.

Chapter 3

Applications of Automata

This chapter will be focused on applications of automata and techniques more or less specific to particular application domains. We will focus on deciding the logics WS1S and WSkS, verification of pointer manipulating programs (shape analysis), string solving, and regular pattern matching.

3.1 Deciding WSkS

Weak monadic second-order logic of k successors (WSkS) is a logic for describing regular properties of finite k-ary trees. WSkS can specify complex properties of trees and also of a rich class of general graphs by referring to their tree backbones [213]. WS1S is the variant restricted to one successor function, the trees are restricted to words. WSkS and WS1S offer extreme succinctness for the price of a non-elementary worst-case complexity. The authors of [147] noticed that through clever implementation techniques and heuristics, the trade-off between complexity and succinctness can be made advantageous. The idea was realized in the tool MONA [133, 185], the best-known implementation of decision procedures for WS1S and WS2S. MONA has found numerous applications in verification of programs with complex dynamic linked data structures [109, 200, 201, 213, 284], string programs [252], array programs [287], parametric systems [69, 75, 89], distributed systems [184, 244], hardware verification [68], automated synthesis [156, 169, 235], and even computational linguistics [214].

On the other hand, despite the extensive research and engineering effort invested into MONA, due to which it still offers the best all-around performance among existing implementations of WS1S/WS2S decision procedures, it is easy to reach its scalability limits. Particularly, MONA implements the classical WS1S/WS2S decision procedure [116] that builds a word/tree automaton representing all models of the formula and then check emptiness of the automaton's language. The automaton is constructed by induction to the formula structure: the construction starts from predefined automata for

atomic predicates and uses automata counterparts of the logical operators to construct an automaton for a formula from the automata for its sub-formulae. The non-elementary worst-case complexity comes from that every quantifier alternation requires determinization, which is exponential. Problematic in practice are also complex boolean combinations of automata. Conjunction and disjunction use the product construction, which is itself quadratic, but a sequence of successive applications of these operations becomes exponential to its length, and it indeed turns out very expensive in practice.

We have been attempting to solve these scalability issues in our works [21, 22, 25]. [21, 22] were inspired by the antichain algorithms [13, 14, 131, 277], discussed in Section 2.3. Recall that the antichain NFA universality test checks language emptiness of the complement automaton *on-the-fly*, while building the complement by the subset construction. The on-the-fly approach allows for significant savings by property-driven pruning—pruning the state space that is irrelevant for the language emptiness test. Antichain algorithms use in fact two forms of pruning. First, *subsumption*, the main optimization, was discussed in Section 2.3. It prunes out states that are "less likely" to lead to a discovery of an accepting state (it basically means disregarding proof obligations that are implied by other ones). Second, *early termination* allows to terminate the search as soon as non-emptiness is proved.

Our decision procedures for WSkS and WS1S generalize the subsumption and early termination discussed above. Essentially, when restricted to quantifier free boolean combinations of atomic predicates, our algorithms search through an implicitly represented state space of the automaton that would be built by the classical decision procedure. The states are represented as *automata terms* combined from states of the atomic automata by boolean connectives, constructed by induction to the input formula. The transition relation between the terms is defined inductively to the terms structure, as well as initial and accepting terms. The main algorithm then searches for an initial term backwards from accepting terms while building the state space of terms on-the-fly. Deciding general quantified formulae then requires to replace a search for an initial state, a simple fixpoint computation, by a nested fixpoint computation, with one nested level for each quantifier alternation. Generalized subsumption (defined inductively to the structure of the terms) is used for pruning as in the antichain algorithms Early termination is generalized into a more interesting form of lazy evaluation of nested fixpoints.

In the case of WS1S and word automata [21], the automata terms may be given a simple language semantics defined inductively based on the languages of the individual states, where the logical operators have their standard set interpretation. This makes proving the correctness of the algorithm almost trivial. Interestingly, in [25] we have not found a direct analogy of this in tree automata. The problem is that the algorithm works in the bottom-up manner, but a single state in a tree automaton does not have a simply defined bottom-up semantics in a form of a language of trees. We therefore had to

prove correctness of the algorithm through defining the meaning of a term more technically, via the classical automata constructions they represent.

We have implemented this approach in several proof of concept tools and compared with MONA. Our tools were able to outperform MONA significantly on certain classes of formulae. It should be said, however, that MONA still remains the bets tool for deciding WSkS overall. One reason is that our tools lack maturity. Another reason is that the on-the-fly approach cannot use classical automata minimization. MONA uses minimization almost after every step, and it is a major tool allowing it to keep automata size at bay. Minimization is however not directly usable in the on-the-fly construction with pruning property driven pruning because it requires the automaton constructed up-front. Combining the on-the-fly and the explicit approach is difficult but it seems promising. We have achieved good results with our first attempt in which we let MONA build minimized automata for sub-formulae (up to certain size of automata or sub-formula depth) and then use our approach to solve the reminder of the formula, with the automata for the sub-formulae playing role of atomic automata.

Our latest work on deciding WSkS is a study on the utility of antiprenexing [24]. It is a well known optimization techniques in SAT/SMT/QBF-solving and theorem proving, essentially trying to restrict the scope of quantifiers by pushing them as deep into the formula as possible. However, it must be used in a controlled way, since an unrestricted use of antiprenexing may enlarge the formula substantially (exponentially in the worst-case). We have therefore designed a heuristic, based on machine learning from data from a number of runs of MONA, to decide whether and which anti-prenexing rules to use (the machine learning is used to estimate the size of an automaton built for a sub-formula). The experiments show that it has a substantial positive impact on the performance of MONA.

Future work. We plan to investigate more sophisticated combinations of classical and on-the-fly approaches, use of abstraction, and techniques for minimization and reduction. One interesting direction is utilising non-deterministic or alternating automata in the explicit automata approach, in the spirit of the non-deterministic automata-based ARTMC [14]. A challenge here is to find a way to complement non-deterministic automata without the need of always determinizing them and to compute projection of an alternating automaton without non-determinizing them. Abstraction in context of WSkS may be used in two flavours. First, the on-the-fly approaches could be combined with ideas from PDR/IC3 [90, 163]. Secondly, the classical approach could be combined with abstraction of the automata structure, such as in [145] and [87]. A new efficient implementation of WSkS might potentially be built on top of a new library of alternating automata, which is one of our primary goals for the nearest future.

Contributed Papers[1]

[21] Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. "Lazy Automata Techniques for WS1S". In: *Proc. of TACAS'17*. Vol. 10205. LNCS. Springer, 2017, pp. 407–425.

[22] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. "Nested Antichains for WS1S". English. In: *Proc. of TACAS'15*. Vol. 9035. LNCS. Springer Berlin Heidelberg, Jan. 2015, pp. 658–674.

[23] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. "Nested antichains for WS1S". In: *Acta Inf.* 56.3 (2019), pp. 205–228.

[24] Vojtěch Havlena, Lukás Holík, Ondrej Lengál, Ondrej Vales, and Tomás Vojnar. "Antiprenexing for WSkS: A Little Goes a Long Way". In: *Proc. of LPAR'20*. Vol. 73. EPiC Series in Computing. EasyChair, 2020, 298–316.

[25] Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. "Automata Terms in a Lazy WSkS Decision Procedure". In: *Proc. of CADE'19*. Vol. 11716. LNCS. Springer, 2019, pp. 300–318.

3.2 Verification of Pointer Programs

Our work on verification of pointer programs (shape analysis) with automata is based on our results with non-deterministic tree automata-based ARTMC. Verification of pointer programs is an opportunity to apply these results in an application domain which is interesting by itself: it is a competitive field and symbolic representation of sets of graphs is a theoretically challenging as well as practically relevant problem.

State of the art in shape analysis. There are many approaches to shape analysis. They are based on logics, e.g., [71, 72, 100, 105, 132, 153, 190, 199, 202, 213, 215, 231, 234, 278, 284], automata [82, 86, 128], graph grammars [157, 272], upward closed sets [52], SMT solving [204], and other formalisms. They differ in their generality, efficiency, and degree of automation. Even though the field as a whole is getting close to practical applications but none of the so far proposed approaches is fully satisfactory.

The leading approach has lately been that of separation logic [231], with the concepts of *separation* and *local reasoning* being its main strengths. Local reasoning stands for the ability of modelling small, local changes of the heap, such as redirecting of a single pointer, through small and local modifications of the abstract representation of the heap. This is achieved through the use of so called separation conjunction that allows to easily express two separated parts of the heap. Local reasoning allows separation logic-based tools such as [72, 190, 278] or [132][2] to achieve relatively impressive scalability. These

[1] The work [25] (in bold) is attached to this thesis.

[2] based on a graph formalism inspired by separation logic

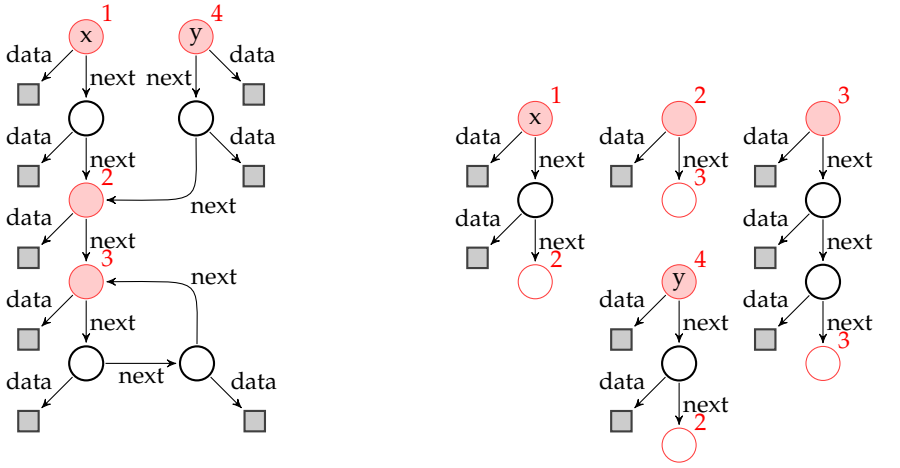
approaches however handle only a restricted class of shape graphs, usually a restricted class of lists, at best also doubly linked, circular, or nested, and especially [72, 278] pay for the scalability by a high imprecision of their analysis. They cannot handle any data structure that cannot be expressed by a fixed set of so called *shape predicates* (inductive descriptions of lists and similar structures). Already trees are mostly beyond their capabilities. Some separation logic-based approaches, namely [153, 189], do handle more complex data structures. They are able to learn inductive predicates automatically. These approaches are however fragile. They seem to be dependent on that the data structures are built in a "nice" way conforming to the structure of the predicate to be learnt (meaning, e.g., that lists are built by adding elements at the end only). The code creating the structure must in fact in a way resemble the desired inductive predicate. [189] for instance fails on examples that might seem easy, e.g., on simple variants of creations and deletions of a doubly-linked list. The principles of learning predicates based on bi-abduction proposed in [189] are however very promising, despite their current shortcomings. Other separation logic approaches that can handle complex data structures, such as [105, 202, 215], are only semi-automated. They require the user to provide the needed inductive predicates, which essentially means providing the shape invariant.

Other less mainstream but mature approach is the grammar-based encoding of heaps from [157, 272], which is conceptually close to our forest automata approach discussed below and also to separation logic (the grammar non-terminal may be loosely compared to inductive predicates of separation logic). It is seldom compared to other approaches, perhaps because it specialises on Java programs, but it seems powerful and viable.

Several older approaches could handle quite complex data strucuters. Successfull was especially TVLA (three valued logic) [196, 234] and earlier automata-based approaches [81, 83, 86]. Their common disadvantage is a monolithic encoding of the heap that does not allow local reasoning, which harms scalability. TVLA also often requires a nontrivial assistance from the user.

Almost all mentioned approaches have difficulties with refining their abstraction and checking spuriousness of counterexamples. Lack of precision and inability to refine the abstraction is often problematic, especially when the needed invariant combines shape properties with properties of the data stored in the heap (e.g. red-black tree manipulations). There are works that try to remedy this, in separation logic [61, 70, 80], [196], a CEGAR-based approach proposed for automated refinement of the *Boolean heap abstraction* [222], the ARTMC-based approaches of [81], or the 2LS approach [204] that leverages SMT solving and bounded model checking.

Our solution discussed below was designed to combine features that do not appear together in any of the approaches discussed above, namely 1) high efficiency, 2) automation, 3) generality and robustness of handled class of shape graphs, and 4) counterexample-based abstraction refinement.

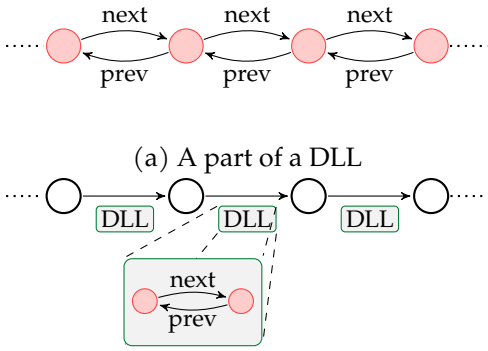


(a) A heap graph with cut-points highlighted in red

(b) The (canonical) tree decomposition of the heap with x ordered before y

Figure 3.1: A heap graph and its tree decomposition

Shape analysis via Forest Automata. Our approach targets verification of sequential C-like programs with complex dynamic pointer data structures such as various forms of singly and doubly-linked lists (SLLs/DLLs), possibly cyclic, shared, hierarchical, and/or having different additional (head, tail, data, and similar) pointers, as well as various forms of trees. In [29], we have proposed an approach of representing sets of heaps via *tree automata*. A heap is split into several *tree components* whose roots are the so-called *cut-points*. Cut-points are nodes pointed to by program variables or having several incoming edges. The tree components can refer to the roots of each other, and hence they are "separated" much like heaps described by formulae joined by the separating conjunction in separation logic. See Figs. 3.1a and 3.1b. Using this decomposition, sets of heaps with a bounded number of cut-points are represented by a new class of automata called *forest automata* (FAs), which are basically tuples of TAs accepting tuples of trees whose leaves can refer back to the roots of the trees. The alphabet symbols used by the FAs may contain *nested FAs*, leading to a *hierarchical encoding of heaps*, so that FAs can represent even sets of heaps with an unbounded number of cut-points (e.g., sets of DLLs). Particularly, a nested FA can describe a part of a heap with a bounded number of cut-points (e.g., a DLL segment). Heaps with an unbounded number of cut-points can be described by using the FA as an alphabet symbol (called *box*) on transitions of other FA. See Figs. 3.2a and 3.2b showing a representation of a DLL (the nested structure may be more complex and can be connected to its environment at more then two points).



(a) A part of a DLL

(b) A hierarchical encoding of the DLL

Figure 3.2: Encoding of a DLL using boxes

FAs appearing in the alphabet of some higher-level FA play a role in some sense similar to that of inductive predicates in separation logic.[3] We however prohibit unbounded recursive nesting and require nesting to form a finite hierarchy. This is obviously different from separation logic where recursive inductive predicates are standard. Instead, we represent this kind of recursion through cycles of tree automata transitions which is easier to handle by finite automata techniques.

We showed in [29] that *entailment* of non-nested FAs (i.e., having a bounded number of cut-points) is decidable. This covers sets of complex structures like SLLs with head/tail pointers. We also showed that entailment can be decided or quite precisely approximated for a large class of nested FAs. Further, C program statements manipulating pointers can be encoded as operations modifying FAs. This made it possible to generalise the essential parts (mainly the forward state space exploration) of the framework of *abstract regular tree model checking* (*ARTMC*) [85, 86] to forest automata and implement a shape analyser Forester based on it. In [34], we improved Forester with automated learning of boxes. It could then automatically verify programs manipulating data structures as complex as two or three level skip-lists. In [33], we gave an algorithm to compute forest automata intersection (under-approximated or even precise for a large class of nested FAs) and implement a generalisation of the counterexample-based abstraction refinement of [85, 86] based on it. This allowed forester to learn complex invariants that combine shape and data properties.

Forest automata in fact combine advantages automata and separation logic. They allow local reasoning similar to that of separation logic. At the same

[3] For instance, we use a nested FA to encode a DLL segment of length 1. In separation logic, the corresponding inductive predicate would represent segments of length 1 or more. In our approach, the repetition of the segment is encoded in the structure of the top-level FA.

time, they inherit higher generality and flexibility of the abstraction of automata approaches, and allow to leverage the recent advances in efficiency of automata technology.

Discussion and Future Directions. FORESTER has automatically verified complex properties of programs with complex data structures such as various flavours of (nested and/or circular) lists, trees, or skip lists. In SV-Comp (Software Verification Competition), it was able to handle a number of benchmarks with complex data structures that no other tool can successfully verify [30–32]. FORESTER was however not meant as a long term software project nor as an all-round capable competitor in SV-Comp. This would require more substantial and systematic implementation effort. Building a more robust tool is one of our future goals. Before that, we are aiming at redesigning the basic formalism of forest automata, in order to remove some of its flaws, such as that forest automata are not closed under union, to get closer to the theoretical limits of decidability of entailment, and to have an overall simpler and cleaner formalism. We are currently trying to develop an new tree automata-based formalism around the ideas of recent works on deciding entailment in separation logic or tree-automata encoded graphs [173, 179]. We wish to utilise such formalism again in a framework inspired by ARMTC, combined with the principle of bi-abduction [100, 189] which promises the ability to handle large and even open code (e.g. analysing a function without the knowledge of its environment), much needed in practice.

Contributed Papers[4]

[26] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. "Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata". In: *Proc. of ATVA'13*. Vol. 8172. LNCS. Springer International Publishing, 2013, pp. 224–239.

[27] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. "Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata". In: *Acta Inf.* 53.4 (2016), pp. 357–385.

[28] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. "Forest Automata for Verification of Heap Manipulation". In: *Proc. of CAV'11*. Vol. 6806. LNCS. Springer Berlin Heidelberg, 2011, pp. 424–440.

[29] **Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. "Forest Automata for Verification of Heap Manipulation". In: *Formal Methods in System Design* 1 (2012), pp. 83–106.**

[4]The work [29] (in bold) is attached to this thesis.

[30] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. "Forester: From Heap Shapes to Automata Predicates - (Competition Contribution)". In: *Proc. of TACAS'17*. Vol. 10206. LNCS. Springer, 2017, 365–369.

[31] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. "Forester: Shape Analysis Using Tree Automata". English. In: *Proceedings of TACAS'15*. Vol. 9035. LNCS. Springer Berlin Heidelberg, Jan. 2015, 432–435.

[32] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. "Run Forester, Run Backwards! - (Competition Contribution)". In: *Proc. of TACAS'16*. Vol. 9636. LNCS. Springer, 2016, 923–926.

[33] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, and Tomáš Vojnar. "Counterexample Validation and Interpolation-Based Refinement for Forest Automata". In: *Proc. of VMCAI'16*. Vol. 10145. LNCS. Springer, 2017, pp. 288–309.

[34] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. "Fully Automated Shape Analysis Based on Forest Automata". In: *Proc. of CAV'13*. Vol. 8044. LNCS. Springer Berlin Heidelberg, 2013, pp. 740–755.

3.3 String Solving

Strings are a fundamental data type in many programming languages. This statement is true now more than ever, especially owing to the rapidly growing popularity of scripting languages (e.g. JavaScript, Python, PHP, and Ruby). String manipulations are often difficult to reason about automatically and could easily lead to unexpected programming errors. Such errors can even have serious security consequences, e.g., cross-site scripting (a.k.a. XSS), ranked among the top three classes of web application security vulnerabilities by OWASP [216].

Popular methods for analysing string manipulations include *symbolic executions* [74, 98, 99, 149, 180, 197, 229, 236, 240] built on top of constraint solvers over the domain of strings (a.k.a. *string solvers*). String solving have been the subject of much research in the past decade or two, e.g., [40, 65, 67, 74, 122, 141–143, 165, 166, 182, 191–194, 236, 259, 260, 268, 271, 279–282, 286]. Most of these works follow the standard approach of *Satisfiability Modulo Theories* (*SMT*) [126] which is an extension of the problem of satisfiability of Boolean formulae wherein each atomic proposition can be interpreted over some logical theories (typically, quantifier-free).

Unlike the case of theories such as integer/real arithmetic (where many decidability and undecidability results are known and powerful algorithms are already available, e.g., the simplex algorithm), string constraints are much less understood. There are many different string operations that can be included in a theory of strings, e.g., concatenation, length comparisons, regular constraints (matching against a regular expression), replace-all (i.e. replac-

ing every occurrence of a string by another string), or conversions between integers and strings. Even for the theory of strings with the concatenation operation alone, existing string solvers cannot handle the theory (in its full generality) in a sound and complete manner, despite the existence of a theoretical decision procedure for the problem [129, 154, 174, 203, 218, 219]. This situation is exacerbated when we add extra operations like string-length comparisons, in which case even decidability is a long-standing open problem [143]. In addition, recent works in string solving have argued in favour of adding the replace-all operator or, more generally, finite-state transducers to string solvers [194, 259, 279, 280] in view of their importance for modelling relevant sanitisers (e.g. backslash-escape) and implicit browser transductions (e.g. an application of HTML-unescape by innerHTML) [122, 165, 268]. However, naively combining the replace-all operator and concatenation yields undecidability [194]. Yet another operation, much needed in application such analysis of scripting languages and yet unexplored, is string-integer conversion.

Among many emerging solutions, automata and transducer-based formalisms seem to be an approach with a good ratio of efficiency, generality, and completeness guarantees [18, 35, 40, 122, 165, 182, 194, 268, 279–281]. Automata are indeed a natural means of representing regular sets of strings. One of the pioneering work on using automata in string solving was our work [40]. It targets string constraints that combine regular constraints, word equation, and Presburger constraints over lengths of strings. In short, word equations are turned into new regular constraints, all (original and new) regular constraints are translated into new constraints on string lengths through computing Parikh images[5] of regular languages, and all (original and new) Presburger constraints are given to an integer arithmetic solver.

A crucial idea is the so called *automata splitting* that eliminates concatenation and transforms the constraints into purely regular ones. As an example, consider the constraint $x.y \in L$. It specifies that the concatenation of the values of string variables x and y belongs to a given regular language L. Is there a satisfying assignment for x and y? (What are the possible assignments?) Automata splitting works as follows. Assuming that L is represented as a finite automaton A, we reason as follows. Every word in L is accepted by a run of A, in which some state q marks the end of the prefix x and beginning of the suffix y. Thus every state q of A represents a class of satisfying assignments where words accepted at q are the x-prefixes and words accepted from q are the y-suffixes. Hence, the entire constraint can be rewritten as a disjunction that has a disjunct $x \in L_q \wedge y \in {}_qL$ for every state q, where L_q is the language of words accepted at q (the automaton A but with the only final state q) and ${}_qL$ is the language of words accepted from q (the automaton A but with three only

[5]The Parikh image of a word assigns to each letter the numbers of its occurrences in the word. The Parikh image of a language is a set of Parikh images of its words. It is usually expressed as an arithmetic formula (semi-linear for regular and context free languages).

initial state q). This eliminates concatenation from the constraint. Automata splitting can be straightforwardly generalised to transducers that encode relations between string variables as shown in [194]. It was then elaborated on in a number of other works [18, 35, 38, 66, 106, 107, 194].

In [18], we propose an efficient implementation of automata splitting that uses alternating automata to encode the large disjunctions produced by the splitting succinctly. Although the alternating automata-based representation means that more expensive satisfiability test, PSPACE-complete namely, its high worst-case complexity can be mitigated by a use of heuristics. We have particularly tried to use the relative recent successful model checking algorithms of IC3/PDR [90, 163] to test emptiness of produced alternating automata. This resulted in a prototype implementation able to handle very complex combinations of transducers and concatenations. Similar attempts were to use model checking methods to solve automata problems made in [119, 270] and even earlier [144]. In our recent paper [39], we have generalised the class of string constraints for which the techniques based on splitting is complete. The resulting *chain-free fragment* of the string constraints logic is one of the largest known decidable logical fragments that combine regular/transducer constraints, concatenation, and length constraints, strictly extending a number of previously proposed fragments (such as the acyclic fragment of [40] or the straight line fragment of [194]) and it seems to precisely capture the limits of the automata splitting.

Our second line of work that started with [35], also starting with that of [40], focuses on using approximation of string constraints that allows their lightweight encoding to Presburger constraints, in effect avoiding expensive automata operations altogether. The idea is that string constraints can be quite faithfully under-approximated by the so called flat automata: automata where simple loops are spawned from a central straight acyclic branch leading from accepting to the final state. The central property of flat automata is that their runs can be precisely characterised by their Parikh images. All main types of string constraints can be under-approximated as flat automata and subsequently turned into arithmetic formulae through a Parikh image construction, including word equations, regular constrains, transducers, or even context-free constraints. The arithmetic formula is then given to an arithmetic SMT solver. The main advantage of this method is that it mostly avoids complex automata construction such as product construction, determinization, or emptiness test of alternating machines, that are needed in the approaches such as [40, 194, 270]. The under-approximation is parameterised by the allowed length of the central branch of the flat automaton and by the length of the attached simple loops. It can be refined by increasing these two parameters. It is complete in the sense that if the string constraint is satisfiable, the solution will be found with some setting of the parameters. The method is very efficient in proving satisfiability. For proving unsatisfiability, it can be paired with any available technique (in our implementation we use [38]). In

our most recent paper [39], we have extended this approach with the ability to handle string-integer constraints that are crucial in applications such as verification of PHP programs. The latest version of our implementation is currently competing very successfully with the best string constraint solvers.

The field of string solving is lively now, partly due its major role in analysing security of web applications. This domain still requires more expressive string constraints handled even more efficiently. There are many avenues of continuing the work discussed here in this direction. Efficient automata techniques, some of which are discussed in this thesis, seem very relevant.

Contributed Papers[6]

[18] **Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. "String constraints with concatenation and transducers solved efficiently". In: *PACMPL* 2.POPL (2018), 4:1–4:32.**

[35] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. "Flatten and conquer: A framework for efficient analysis of string constraints". In: *Proc. of PLDI'17*. ACM Trans. Comput. Log., 2017, 602–617.

[36] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukáš Holík, Ahmed Rezine, and Philipp Rümmer. "Trau: SMT solver for string constraints". In: *Proc. of FMCAD'18*. IEEE, 2018, pp. 1–5.

[37] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. "Norn: An SMT Solver for String Constraints". In: *Proc. of CAV'15*. Vol. 9206. LNCS. Springer, 2015, pp. 462–469.

[38] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukáš Holík, and Petr Janků. "Chain-Free String Constraints". In: *Proc. of ATVA'19*. Vol. 11781. LNCS. Springer, 2019, pp. 277–293.

[39] Parosh Aziz Abdulla et al. "Efficient handling of string-number conversion". In: *Proc. of PLDI'20*. ACM, 2020, pp. 943–957.

[40] **ParoshAziz Abdulla, MohamedFaouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. "String Constraints for Verification". English. In: *Proc. of CAV'14*. Vol. 8559. LNCS. Springer International Publishing, 2014, pp. 150–166.**

3.4 Pattern Matching

Matching regular expressions is a ubiquitous component of computer systems used e.g. for searching, data validation, parsing, find/replace, data scrapping, syntax highlighting, and it is a commonly used and natively supported in most programming languages [117]. About 30-40% of Java, JavaScrip, and

[6] The works [18, 40] (in bold) are attached to this thesis.

Python software uses regex matching (as reported in multiple studies, e.g. [124]). Network Intrusion Detection Systems (IDSes) use hardware accelerated pattern matching. Frameworks such as Snort [198], Suricata [208], or Bro [232], are widely used to secure internet connection against attacks and malicious traffic.

We will discuss two lines of work on pattern matching, one more relevant to pattern matching of software, one more relevant to pattern matching of hardware. The challenges we are addressing are specific to the given context, therefore we discuss the two separately.

Counting in Software Pattern Matching

This line of work aims at improving efficiency of software regular pattern matching for the particular case of regexes with bounded repetition, which are often problematic for existing matchers.

Efficiency of matching engines have a significant impact on the overall usability of software applications. Unpredictability of a matchers performance may lead to catastrophic consequences, witnessed by events such as the recent catastrophic outage of Cloudfare services [152], caused by a single poorly written regex, and it is a doorway for the so called ReDOS attack, a denial of service attack based on overwhelming the matching engine by providing a specially crafted regex or text. In 2016 ReDoS caused for instance an outage at StackOverflow [138], or rendered vulnerable websites that used the popular Express.js framework [64]. Also works such as [124, 125] argue that ReDOS is not just a niche problem, but rather a common and serious thread.

Failures of matching are mostly caused by "catastrophic backtracking", a situation when variants of Spencer's simulation of an NFA by backtracking [247] exhibits a behaviour super-linear to the length of the text. Spencer style algorithms are implemented most often, though they are at worst exponential in the text length. An alternative with a much lower worst-case complexity in the text size is to use deterministic automata (DFA). In the ideal case, the DFA is pre-computed. Matching can then be linear in text length, constant in processing each character. This is the so called static DFA simulation [242]. The major drawback of this approach is that the determinization of NFA may explode, rendering the method unusable. Variants of Thompson's NFA-to-DFA simulation [256], where the deterministic automaton is built on-the-fly, reduce the impact of exploding determinization. Naive implementation of Thompson's algorithm would have to pay a hefty price for each matched character (at most linear to the size of the NFA, one step of the on-the-fly subset construction may use all transitions and states of the NFA), but the modern implementations, such as [148, 150, 269] use caching of the already visited DFA states to achieve high efficiency (even though even they are completely immune to exploding implementation).

In our works [19, 20], we focus on eliminating a frequent cause of DFA explosion—a use of the *counting operator* also known as the operator of *limited repetition*. It can succinctly express patterns such as `(ab){1,100}` representing words where `ab` appears 1 to 100 times. Such expressions are very common (cf. [79]), e.g., in the RegExLib library [230], which collects expressions for recognising URIs, markup code, pieces of Java code, or SQL queries; in the Snort rules [198] used for finding attacks in network traffic; in real-life XML schemas, with the counter bounds being as large as 10 million [79]; or in detecting information leakage from traffic logs [19].

To illustrate the principal difficulty with matching of bounded repetitions, consider the regex `.*a.{`k`}` where $k \in \mathbb{N}$ (`a` appears k-characters from the end). The minimal DFA accepting the language has 2^{k+1} states because it must remember all the positions where `a` was seen during the last $k+1$ steps. This requires a finite memory of $k+1$ bits and thus 2^{k+1} reachable DFA states. Determinizing explicitly is hence out of question. The Thompson style algorithms, using on the on-the-fly determinization, will also run into problems since processing nearly every character will require generating a new DFA state, represented as set of NFA states of the size up to k.

As a cure to this problem, we have proposed in [20] to use a translation from regexes with repetition to small deterministic machines that can be simulated in matching with nearly constant character complexity. Our compilation from regexes to CsA proceeds in two steps. First, we compile the regexes into non-deterministic *counting automata* (*CA*), automata with bounded counters. Variants of CA have been used in several other works under different names e.g. [19, 168, 183, 245, 248] all essentially boiling down to variations on counter automata with counters limited to a bounded range of values. The compilation from regular expressions is cheap and produces counting automata with the size independent of the counter bounds. The major technical problem is then *determinization of the CA* in a way that would not explode in the counter bounds and would ideally produce deterministic machines of a size independent of the counter bounds.Our first solution, published in [19], was to produce a deterministic CA which would for a practically significant class of regexes be of a size *linear to the counter bounds*. [20] improves on this and presents an algorithm that produces deterministic a *counting set automata* (CsA) in time *independent of the counter bounds*.

The idea of the CA to CsA determinization of [20] is best explained by comparison with the naive determinization of CA which creates a DFA by explicit subset construction. The states of the DFA are sets of runtime configurations of the CA: each CA-configuration consisting of a control state and a counter valuation. Counter valuations are hence "unfolded", they become an explicit part of the DFA control states, and the succinctness provided by counters is completely lost. In contrast to this, our CsA represents the counter valuations implicitly: it computes them dynamically on-the-fly and stores them in the counting sets. To do that, CsA are equipped with a special type of reg-

isters, called *counting sets*, that can hold a set of bounded integer values and support a limited set of simple set operations that can all be implemented to run in constant time (add 0, add 1, empty the set, increase all, compare the maximal element with a constant). The counter valuations are hence not a part a control state and their overall number influences neither the size of the CsA nor the time needed to build them. Even though our determinization so far does not work for the full class of regular expressions with counting, the supported class of regexes is general enough to handle the absolute majority of regular expressions found in practice. Our experimental results from [20] confirm that CsA are able to eliminate most problems that modern regexes, such as `grep` [148], RE2 [150], and [269] have with bounded counting.

This work has many promising future directions. Matching would utilise efficient boolean operations over CsA and means of minimization/size reduction. CsA also have a good potential for the use as a symbolic representation of sets. Applications may arise from frameworks for verification of programs, such as [158], where automata represent the control flow and a bounded for-loop induces an automaton with counting. Other applications may appear in solving WS1S formulas or Presburger formulas with automata, in string solving where constraints over string lengths are common and can be encoded in automata using counting. A challenging open problem is an emptiness check for this kind of automata with a run-time ideally independent of the counter bounds, it is needed in all these application domains.

Approximation in Hardware Accelerated Regex Matching

The work discussed here aims at hardware accelerated regex matching, namely, at minimization of hardware resources needed to realize matching of large and complicated patterns in network monitoring.

Intrusion Detection Systems (IDSes), such as Snort [198], Suricata [208], or Bro [232] are widely used to secure Internet connection against attacks and malicious traffic. One of the prominent approaches for IDSes is *deep packet inspection* (DPI), which is based on matching regular expressions describing attack patterns against network traffic.

The networks operate at impressive and still growing speeds—telecommunication companies started to deploy 100 Gbps links, the 400 Gbps Ethernet standard has recently been ratified [170], and large data centers already call for a 1 Tbps technology. Existing IDSes struggle. The best software-based solutions, such as [265], can achieve a 100 Gbps throughput using Bro on a cluster of servers. A single-box IDS is at such speeds far beyond the capabilities of software-based solutions—hardware acceleration is needed.

A technology increasingly used in data centers [101, 224] that provides provide high computing power and flexibility is that of *field-programmable gate arrays* (FPGAs). They achieve matching speeds over 100 Gbps [207], however, only for the price of massive parallelization that puts excessive demands

on the resources of FPGAs. For instance, in the HW architecture that we propose, processing 400 Gbps input network traffic requires 256 concurrently functioning regex matching units. Reducing the consumed resources is thus of paramount importance.

The FPGA matching units implement either deterministic or non-deterministic finite automata (hardware parallelization allows efficient implementation of the Thomposon's NFA simulation [256]). In our works [41, 103], we focus on NFAs since they take much less resources on the FPGA, as shown, e.g., by [110, 195, 241, 246, 283]. We propose a method of reducing the sizes of NFAs even for the price of over-approximating the language, by techniques that provide good trade-offs between the precision and reduction factors and overall much greater reduction then known exact techniques (such as simulation reduction). Subsequently, we propose a *multi-stage* architecture [41] of the regex matching engine. Consider an NFA A that recognizes the language L of a given set of regexes. The proposed architecture is composed of several stages where each subsequent stage uses more precise and hence larger automata, but requires less parallelization as it receives traffic filtered by the previous less precise but faster and more parallelized stage.

The first version of our approximate reduction techniques [103] was based on learning probabilistic model of traffic in a form of a probabilistic automaton, computing its product with the automaton to be minimized, and pruning and merging insignificant states. This method gives some interesting theoretical guarantees but is rather expensive in practice. Therefore, in [41] we use a more lightweight approach. Given an NFA constructed from the regexes of interest, we label its states with their *significance*—the likelihood that they will be used during processing a packet obtained by running a sample of the network traffic trough the automaton and counting how many times each state was used–and then simplify the least significant parts of the automaton. The simplification is again implemented by *pruning* and *merging* of the insignificant states. This methods scales well. On regexes taken from the IDS Snort and other resources, it yielded a substantial reduction of the size of the NFAs while keeping the number of false positives low. When used within the multi-stage architecture, it enabled to perform matching at 100 and 400 Gbps on sets of regexes whose sizes were far beyond the capabilities of previous solutions.

We plan to continue this work by exploring other means of approximate automata state space reduction, such as those inspired by classical simulation-based reductions, perhaps close to probabilistic simulations [127]. We will also have a closer look into the process of translating automata to FPGA hardware and analyse opportunities for better integration with the approximate reductions there.

Chapter 4

Concurrent Systems

This chapter discusses research on extensions of automata with parallelism, Petri nets and their generalizations, with applications in verification of parameterized systems and parallel pointer programs.

We started this direction with a work on verification framework for lock-free implementations of pointer data structures [45]. To verify linearisability of lock-free data structures such as queues or stacks, we combined three components: 1) thread-modular reasoning to handle parallelism, 2) a specification mechanism based on finite automata observer to specify the behaviour of locks and stacks (e.g. the FIFO or the LIFO property), and 3) a simple abstract domain specialised to capture configurations of parallel programs manipulating singly linked lists. One of the highlights of the work was the idea of using automata to specify essentially non-regular behaviours of stacks and queues. We showed that under the assumption of data independence, adapted from [274] (essentially, the program does not look at the data stored in the data structure), the violations of the stack or queue-like behaviours may be captured by regular properties over the alphabet of simple events (such ass $insert(a)$, $delete(b)$, where a and b are one of about three abstract data values).

The main obstacle turned out to be the insufficient precision of thread-modular abstraction which we were using to capture unbounded parallelism. Namely, thread-modular abstraction abstracts a state of a program with arbitrarily many threads into a set of local states of the individual threads. In the case of heap configurations, the local state would refer roughly to the part of the heap reachable from the local variables of the thread and from global variables. Reasoning about programs such as Michael and Scot's queue [211] under thread-modular abstraction however produces false positives.

On the way to remedy this, we discovered that the precision of thread-modular abstraction may be improved in a parametric manner. Simply said, instead of remembering a local state of one thread, one can remember a local state every k-tuple of threads, for some constant k which is a parameter of the abstract domain. We call these k-tuples *views*. The higher the k, the

more precision the abstract domain has. Interestingly, as we show in [43], general coverability of well-structured systems (e.g. Petri nets or, broadcast communication protocols, lossy channels) can always be verified with a high enough k. This gives a complete algorithm for checking coverability in well-structured systems: run the system with increasing values of k until either it is proven safe or a real counterexample trace is discovered. Among modern approaches and heuristics for coverability in Petri nets and well-structured transition systems [42, 51, 146, 178, 186, 249], view abstraction provides very good compromise between efficiency and simplicity. It makes it easy to generalise to other, more complex, systems. With our prototype implementation, we were able to verify, besides a number of Petri net models, models of communication protocols with various complex topologies. The notion of a view easily generalises from a tuple of a threads to e.g. a limited sub-graph of the topology. In [47], we further extended the abstract domain of views with so called contexts which led to a verification algorithm complete even beyond well-structured systems. Intuitively, views represent an universal property: all views within a program state must satisfy it. The contexts on the other hand represent an existential property: if a view appears in the global state, then the global state must have also some other property. This allowed to verify certain non-monotonic systems such as e.g. the full Szymanski's mutual exclusion protocol (to the best of our knowledge, there is still no other automatic method capable of verifying Szymanski's protocol).

The work on verification of linearisability properties could then be completed using the technique of [43] which allowed to refine the thread-modular abstraction sufficiently. Namely, we could verify basic lock-free data structures such as Treiber's stack [258] and Michael and Scott's queue [211], and the method delivered overall comparable or better performance and precision then the closest work [261, 263] based on separation logic and assume guarantee reasoning.

We have continued this work in [48, 49] where we studied the same problem, verification of linearizability of stacks and queues, but with an emphasis on achieving good scalability under the absence of garbage collector. In this situation, the so called ABA problem may cause that a program correct under garbage collection becomes incorrect due to the possibility of an accidental reuse of previously freed memory. Verification of absence of ABA related bugs under the absence of garbage collector requires substantially more precision, which in turn induces very high computational demands. We have succeeded in devising techniques for proving such programs correct with relatively minimal additional costs. Namely, in [48], we have proposed a notions of *pointer races*, an analogue of the classical notion of race, and shown that verification can be divided into two easier sub-tasks: first, show that absence of *strong pointer races*, second, under the assumption of strong pointer race freeness, verify the original property of interest under the much less demanding garbage collecting semantics. Further, in [49] we proposed a tech-

nique of so called summaries to limit the cost of the abstract transformer in the thread-modular abstract domain. Roughly, the original code is replaced with so called *summaries*, pieces of atomic and "stateless" code that overapproximates the original program. The abstract post operator then becomes much cheaper. The verification is again split into two simpler tasks, verifying that the overapproximation is sound, and verify that the overapproximation preserves the original property of interest. This considerably improves scalability of the entire method.

Our work on parallel pointer program verification lies on a crossroad of the field of shape analysis and analysis of parallelism, and it is relevant to a wide context of related works. There have been many works on thread-modular and assume guarantee, with a general focus as well as focused on pointer programs [73, 139, 151, 175, 205, 239, 262–264]. Other techniques for parallel program verification include atomicity abstraction [50, 135, 136, 176, 223, 233] relevant especially to [49] and simulation [130, 135, 237, 285]. We omit the entire rich field of sequential pointer program analysis, which is discussed more in Section 3.2. Our original work [45] was very close to the works of Vafeiadis [262–264] who obtained similar results using assume-guarantee reasoning in separation logic. Our works [48, 49] are still among the most successful in verifying pointer programs with complex low level synchronisation, together with several follow-ups of our colleagues [58, 210] which elaborated on some of their aspects further.

Contributed Papers[1]

[42] **Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. "All for the Price of Few". In: *Proc. of VMCAI'13*. Vol. 7737. LNCS. Springer Berlin Heidelberg, 2013, pp. 476–495.**

[43] Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. "Parameterized Verification through View Abstraction". In: *Int. J. Softw. Tools Technol. Transf.* 18.5 (2016), 495–516.

[44] Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. "View Abstraction - A Tutorial (Invited Paper)". In: *Proc. of SynCoP'15*. Vol. 44. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 1–15.

[45] Parosh Aziz Abdulla, Frédéric Haziza, Lukáš Holík, Bengt Jonsson, and Ahmed Rezine. "An Integrated Specification and Verification Technique for Highly Concurrent Data Structures". In: *Proc. of TACAS'13*. Vol. 7795. LNCS. Springer Berlin Heidelberg, 2013, pp. 324–338.

[46] **Parosh Aziz Abdulla, Frédéric Haziza, Lukáš Holík, Bengt Jonsson, and Ahmed Rezine. "An Integrated Specification and Verification Technique for Highly Concurrent Data Structures". In: *Int. J. Softw. Tools Technol. Transf.* 19.5 (2017), pp. 549–563.**

[1] The works [42, 46] (in bold) are attached to this thesis.

[47] ParoshAziz Abdulla, Frédéric Haziza, and Lukáš Holík. "Block Me If You Can!" English. In: *Proc. of SAS'14*. Vol. 8723. LNCS. Springer International Publishing, Jan. 2014, pp. 1–17.

[48] Frédéric Haziza, Lukáš Holík, Roland Meyer, and Sebastian Wolff. "Pointer Race Freedom". In: *Proc. of VMCAI'16*. Vol. 9583. LNCS. Springer, 2016, pp. 393–412.

[49] Lukáš Holík, Roland Meyer, Tomáš Vojnar, and Sebastian Wolff. "Effect Summaries for Thread-Modular Analysis - Sound Analysis Despite an Unsound Heuristic". In: *Proc. of SAS'17*. Vol. 10422. LNCS. Springer, 2017, 169–191.

Bibliography

[1] P. A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. "Composed Bisimulation for Tree Automata". In: *Proc. of CIAA'08*. Vol. 5148. LNCS. Springer Berlin Heidelberg, 2008, pp. 212–222.

[2] P. A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. "Computing Simulations over Tree Automata". In: *Proc. of TACAS'08*. Vol. 4963. LNCS. Springer Berlin Heidelberg, 2008, pp. 93–108.

[3] P. A. Abdulla, Y.-F. Chen, L. Holík, and T. Vojnar. "Mediating for Reduction (on Minimizing Alternating Büchi Automata)". In: *Proc. of FSTTCS'09*. Vol. 4. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2009, 1–12.

[4] P. A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. "Composed Bisimulation for Tree Automata". In: *International Journal of Foundations of Computer Science* 04 (2009), pp. 685–700.

[5] P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. "Simulation Subsumption in Ramsey-Based Büchi Automata Universality and Inclusion Testing". In: *Proc. of CAV'10*. Vol. 6174. LNCS. Springer Berlin Heidelberg, 2010, pp. 132–147.

[6] P. A. Abdulla, Y.-F. Chen, L. Holík, and T. Vojnar. "Mediating for reduction (on minimizing alternating Büchi automata)". In: *Theoretical Computer Science* 552.0 (Oct. 2014), pp. 26–43.

[7] P. A. Abdulla, L. Holík, L. Kaati, and T. Vojnar. "A Uniform (Bi-) Simulation-Based Framework for Reducing Tree Automata". In: *Electron. Notes theor. Comput. Sci.* 251 (2009), pp. 27–48.

[8] R. Almeida, L. Holík, and R. Mayr. "Reduction of Nondeterministic Tree Automata". In: *Proc. of TACAS'16*. Vol. 9636. LNCS. Springer, 2016, 717–735.

[9] L. Holík. "Simulations and Antichains for Efficient Handling of Finite Automata". PhD thesis. Brno, CZ: Department of Intelligent Systems FIT BUT, 2011.

[10] L. Holík and J. Šimáček. "Optimizing an LTS-Simulation Algorithm". In: *Proc. of MEMICS'09*. Znojmo, CZ: Faculty of Informatics MU, 2009, 93–101.

[11] L. Holík, O. Lengál, J. Síč, M. Veanes, and T. Vojnar. "Simulation Algorithms for Symbolic Automata". In: *Proceedings of ATVA'18*. Vol. 11138. LNCS. Springer, 2018, pp. 109–125.

[12] P. A. Abdulla, Y.-F. Chen, L. Clemente, L. Holík, C.-D. Hong, R. Mayr, and T. Vojnar. "Advanced Ramsey-Based Büchi Automata Inclusion Testing". In: *Proc. of CONCUR'11*. Vol. 6901. LNCS. Springer Berlin Heidelberg, 2011, pp. 187–202.

[13] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. "When Simulation Meets Antichains". In: *Proc. of TACAS'10*. Vol. 6015. LNCS. Springer Berlin Heidelberg, 2010, pp. 158–174.

[14] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. "Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata". In: *Proc. of CIAA'08*. Vol. 5148. LNCS. Springer Berlin Heidelberg, 2008, pp. 57–67.

[15] L. Holík, O. Lengál, J. Šimáček, and T. Vojnar. "Efficient Inclusion Checking on Explicit and Semi-symbolic Tree Automata". In: *Proc. of ATVA'11*. Vol. 6996. LNCS. Springer Berlin Heidelberg, 2011, pp. 243–258.

[16] L. Holík and R. Meyer. "Antichains for the Verification of Recursive Programs". In: *Proc. of NETYS'15*. Vol. 9466. LNCS. Springer, 2015, 322–336.

[17] L. Holík, R. Meyer, and S. Muskalla. "Summaries for Context-Free Games". In: *Proc. of FSTTCS'16*. Vol. 65. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 41:1–41:16.

[18] L. Holík, P. Janků, A. W. Lin, P. Rümmer, and T. Vojnar. "String constraints with concatenation and transducers solved efficiently". In: *PACMPL* 2.POPL (2018), 4:1–4:32.

[19] L. Holík, O. Lengál, O. Saarikivi, L. Turoňová, M. Veanes, and T. Vojnar. "Succinct Determinisation of Counting Automata via Sphere Construction". In: *Proc. of APLAS'19*. Vol. 11893. LNCS. Springer, 2019, pp. 468–489.

[20] L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar. *Regex Matching with Counting-Set Automata*. accepted at OOPSLA'20. 2020.

[21] T. Fiedor, L. Holík, P. Janků, O. Lengál, and T. Vojnar. "Lazy Automata Techniques for WS1S". In: *Proc. of TACAS'17*. Vol. 10205. LNCS. Springer, 2017, pp. 407–425.

[22] T. Fiedor, L. Holík, O. Lengál, and T. Vojnar. "Nested Antichains for WS1S". English. In: *Proc. of TACAS'15*. Vol. 9035. LNCS. Springer Berlin Heidelberg, Jan. 2015, pp. 658–674.

[23] T. Fiedor, L. Holík, O. Lengál, and T. Vojnar. "Nested antichains for WS1S". In: *Acta Inf.* 56.3 (2019), pp. 205–228.

[24] V. Havlena, L. Holík, O. Lengál, O. Vales, and T. Vojnar. "Antiprenexing for WSkS: A Little Goes a Long Way". In: *Proc. of LPAR'20*. Vol. 73. EPiC Series in Computing. EasyChair, 2020, 298–316.

[25] V. Havlena, L. Holík, O. Lengál, and T. Vojnar. "Automata Terms in a Lazy WSkS Decision Procedure". In: *Proc. of CADE'19*. Vol. 11716. LNCS. Springer, 2019, pp. 300–318.

[26] P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar. "Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata". In: *Proc. of ATVA'13*. Vol. 8172. LNCS. Springer International Publishing, 2013, pp. 224–239.

[27] P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar. "Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata". In: *Acta Inf.* 53.4 (2016), pp. 357–385.

[28] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. "Forest Automata for Verification of Heap Manipulation". In: *Proc. of CAV'11*. Vol. 6806. LNCS. Springer Berlin Heidelberg, 2011, pp. 424–440.

[29] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. "Forest Automata for Verification of Heap Manipulation". In: *Formal Methods in System Design* 1 (2012), pp. 83–106.

[30] L. Holík, M. Hruška, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. "Forester: From Heap Shapes to Automata Predicates - (Competition Contribution)". In: *Proc. of TACAS'17*. Vol. 10206. LNCS. Springer, 2017, 365–369.

[31] L. Holík, M. Hruška, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. "Forester: Shape Analysis Using Tree Automata". English. In: *Proceedings of TACAS'15*. Vol. 9035. LNCS. Springer Berlin Heidelberg, Jan. 2015, 432–435.

[32] L. Holík, M. Hruška, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. "Run Forester, Run Backwards! - (Competition Contribution)". In: *Proc. of TACAS'16*. Vol. 9636. LNCS. Springer, 2016, 923–926.

[33] L. Holík, M. Hruška, O. Lengál, A. Rogalewicz, and T. Vojnar. "Counterexample Validation and Interpolation-Based Refinement for Forest Automata". In: *Proc. of VMCAI'16*. Vol. 10145. LNCS. Springer, 2017, pp. 288–309.

[34] L. Holík, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. "Fully Automated Shape Analysis Based on Forest Automata". In: *Proc. of CAV'13*. Vol. 8044. LNCS. Springer Berlin Heidelberg, 2013, pp. 740–755.

[35] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. "Flatten and conquer: A framework for efficient analysis of string constraints". In: *Proc. of PLDI'17*. ACM Trans. Comput. Log., 2017, 602–617.

[36] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. "Trau: SMT solver for string constraints". In: *Proc. of FMCAD'18*. IEEE, 2018, pp. 1–5.

[37] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. "Norn: An SMT Solver for String Constraints". In: *Proc. of CAV'15*. Vol. 9206. LNCS. Springer, 2015, pp. 462–469.

[38] P. A. Abdulla, M. F. Atig, B. P. Diep, L. Holík, and P. Janků. "Chain-Free String Constraints". In: *Proc. of ATVA'19*. Vol. 11781. LNCS. Springer, 2019, pp. 277–293.

[39] P. A. Abdulla et al. "Efficient handling of string-number conversion". In: *Proc. of PLDI'20*. ACM, 2020, pp. 943–957.

[40] P. Abdulla, M. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. "String Constraints for Verification". English. In: *Proc. of CAV'14*. Vol. 8559. LNCS. Springer International Publishing, 2014, pp. 150–166.

[41] M. Ceska et al. "Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata". In: *Proc. of FCCM'19*. IEEE, 2019, pp. 109–117.

[42] P. A. Abdulla, F. Haziza, and L. Holík. "All for the Price of Few". In: *Proc. of VMCAI'13*. Vol. 7737. LNCS. Springer Berlin Heidelberg, 2013, pp. 476–495.

[43] P. A. Abdulla, F. Haziza, and L. Holík. "Parameterized Verification through View Abstraction". In: *Int. J. Softw. Tools Technol. Transf.* 18.5 (2016), 495–516.

[44] P. A. Abdulla, F. Haziza, and L. Holík. "View Abstraction - A Tutorial (Invited Paper)". In: *Proc. of SynCoP'15*. Vol. 44. OASICS. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 1–15.

[45] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. "An Integrated Specification and Verification Technique for Highly Concurrent Data Structures". In: *Proc. of TACAS'13*. Vol. 7795. LNCS. Springer Berlin Heidelberg, 2013, pp. 324–338.

[46] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. "An Integrated Specification and Verification Technique for Highly Concurrent Data Structures". In: *Int. J. Softw. Tools Technol. Transf.* 19.5 (2017), pp. 549–563.

[47] P. Abdulla, F. Haziza, and L. Holík. "Block Me If You Can!" English. In: *Proc. of SAS'14*. Vol. 8723. LNCS. Springer International Publishing, Jan. 2014, pp. 1–17.

[48] F. Haziza, L. Holík, R. Meyer, and S. Wolff. "Pointer Race Freedom". In: *Proc. of VMCAI'16*. Vol. 9583. LNCS. Springer, 2016, pp. 393–412.

[49] L. Holík, R. Meyer, T. Vojnar, and S. Wolff. "Effect Summaries for Thread-Modular Analysis - Sound Analysis Despite an Unsound Heuristic". In: *Proc. of SAS'17*. Vol. 10422. LNCS. Springer, 2017, 169–191.

[50] M. Abadi and L. Lamport. "The Existence of Refinement Mappings". In: *Proc. of LICS'88*. IEEE Computer Society, 1988, pp. 165–175.

[51] P. A. Abdulla. "Well (and better) quasi-ordered transition systems". In: *Bulletin of Symbolic Logic* 16.4 (2010), pp. 457–515.

[52] P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. "Monotonic Abstraction for Programs with Dynamic Memory Heaps". In: *Proc. of CAV'08*. Vol. 5123. LNCS. Springer, 2008, pp. 341–354.

[53] P. A. Abdulla, G. Delzanno, and M. Montali. "Well Structured Transition Systems with History". In: *Proc. of GandALF'15*. Vol. 193. EPTCS. 2015, pp. 115–128.

[54] P. A. Abdulla, J. Högberg, and L. Kaati. "Bisimulation Minimization of Tree Automata". In: *Int. J. Found. Comput. Sci.* 18.4 (2007), pp. 699–713.

[55] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. "Algorithmic Improvements in Regular Model Checking". In: *Proc. of CAV'03*. Vol. 2725. LNCS. Springer, 2003, pp. 236–248.

[56] P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. "Regular Model Checking for LTL(MSO)". In: *Proc. of CAV'04*. Vol. 3114. LNCS. Springer, 2004, pp. 348–360.

[57] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. "A Survey of Regular Model Checking". In: *Proc. of CONCUR'04*. Vol. 3170. LNCS. Springer, 2004, pp. 35–48.

[58] P. A. Abdulla, B. Jonsson, and C. Q. Trinh. "Automated Verification of Linearization Policies". In: *Proc. of SAS'16*. Vol. 9837. Springer, 2016, 61–83.

[59] P. A. Abdulla, A. Legay, J. d'Orso, and A. Rezine. "Tree Regular Model Checking: A Simulation-Based Approach". In: *J. Log. Algebr. Program.* 69.1-2 (2006), pp. 93–121.

[60] A. V. Aho and J. D. Ullman. "Translations on a Context-Free Grammar". In: *Inf. Comput.* 19.5 (1971), pp. 439–475.

[61] A. Albarghouthi, J. Berdine, B. Cook, and Z. Kincaid. "Spatial Interpolants". In: *Proc. of ESOP'15*. Vol. 9032. LNCS. Springer, 2015.

[62] R. Alur and M. Yannakakis. "Model Checking of Message Sequence Charts". In: *Proc. of CONCUR'99*. Vol. 1664. LNCS. Springer, 1999, pp. 114–129.

[63] A. Avron, N. Dershowitz, and A. Rabinovich. "Boris A. Trakhtenbrot: Academic Genealogy and Publications". In: *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Springer, 2008, pp. 46–57.

[64] A. Baldwin. *Regular Expression Denial of Service affecting Express.js*. 2016.

[65] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications". In: *Proc. of S&P'08*. IEEE Computer Society, 2008, pp. 387–401.

[66] P. Barceló, C. Hong, X. B. Le, A. W. Lin, and R. Niskanen. "Monadic Decomposability of Regular Relations". In: *Proc. of ICALP'19*. Vol. 132. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 103:1–103:14.

[67] C. W. Barrett, C. Tinelli, M. Deters, T. Liang, A. Reynolds, and N. Tsiskaridze. "Efficient solving of string constraints for security analysis". In: *Proc. of HotSoS'16*. ACM Trans. Comput. Log., 2016, pp. 4–6.

[68] D. Basin and N. Klarlund. "Automata Based Symbolic Reasoning in Hardware Verification". In: *Proc. of CAV'98*. LNCS. Springer, 1998, pp. 349–361.

[69] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. "Abstracting WS1S Systems to Verify Parameterized Networks". In: *Proc. of TACAS'00*. Vol. 1785. LNCS. Springer, 2000, pp. 188–203.

[70] J. Berdine, A. Cox, S. Ishtiaq, and C. Wintersteiger. "Diagnosing Abstraction Failure for Separation Logic-based Analyses". In: *Proc. of CAV'12*. Vol. 7358. LNCS. Springer, 2012.

[71] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. "Shape Analysis for Composite Data Structures". In: *Proc. of CAV'07*. Vol. 4590. LNCS. Springer, 2007, pp. 178–192.

[72] J. Berdine, B. Cook, and S. Ishtiaq. "SLAyer: Memory Safety for Systems-Level Code". In: *Proc. of CAV'11*. Vol. 6806. LNCS. Springer, 2011, 178–183.

[73] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. "Thread Quantification for Concurrent Shape Analysis". In: *Proc. of CAV'08*. Vol. 5123. LNCS. Springer, 2008, pp. 399–413.

[74] N. Bjørner, N. Tillmann, and A. Voronkov. "Path Feasibility Analysis for String-Manipulating Programs". In: *Proc. of TACAS'09*. Vol. 5505. LNCS. Springer, 2009, pp. 307–321.

[75] J. Bodeveix and M. Filali. "FMona: A Tool for Expressing Validation Techniques over Infinite State Systems". In: *Proc. of ETAPS'00*. Vol. 1785. LNCS. Springer, 2000, pp. 204–219.

[76] B. Boigelot, A. Legay, and P. Wolper. "Iterating Transducers in the Large (Extended Abstract)". In: *Proc. of CAV'03*. Vol. 2725. LNCS. Springer, 2003, pp. 223–235.

[77] B. Boigelot and P. Wolper. "Representing Arithmetic Constraints with Finite Automata: An Overview". In: *Proc. of ICLP'02*. Vol. 2401. LNCS. Springer, 2002, pp. 1–19.

[78] F. Bonchi and D. Pous. "Checking NFA equivalence with bisimulations up to congruence". In: *Proc. of POPL'13*. ACM, 2013, pp. 457–468.

[79] E. Börklund, W. Martens, and T. Timm. "Efficient Incremental Evaluation of Succinct Regular Expressions". In: *Proc. of CIKM'15*. ACM. 2015.

[80] M. Botincan, M. Dodds, and S. Magill. "Refining Existential Properties in Separation Logic Analyses". In: *CoRR* abs/1504.08309 (2015). arXiv: 1504.08309.

[81] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. "Abstract Regular (Tree) Model Checking". In: *International Journal on Software Tools for Technology Transfer* 14.2 (2012), pp. 167–191.

[82] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. "Programs with Lists are Counter Automata". In: *Formal Methods in System Design* 38.2 (2011), pp. 158–192.

[83] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. "Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking". In: *Proc. of TACAS'05*. Vol. 3440. LNCS. Springer, 2005, pp. 13–29.

[84] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. "Abstract regular (tree) model checking". In: *Int. J. Softw. Tools Technol. Transf.* 14.2 (2012), pp. 167–191.

[85] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. "Abstract Regular Tree Model Checking". In: *Electr. Notes Theor. Comput. Sci.* 149.1 (2006), pp. 37–48.

[86] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. "Abstract Regular Tree Model Checking of Complex Dynamic Data Structures". In: *Proc. of SAS'06*. Vol. 4134. LNCS. Springer, 2006, pp. 52–70.

[87] A. Bouajjani, P. Habermehl, and T. Vojnar. "Abstract Regular Model Checking". In: *Proc. of CAV'04*. Vol. 3114. LNCS. Springer, 2004, pp. 372–386.

[88] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. "Regular Model Checking". In: *Proc. of CAV'00*. Vol. 1855. LNCS. Springer, 2000, pp. 403–418.

[89] M. Bozga, R. Iosif, and J. Sifakis. "Structural Invariants for Parametric Verification of Systems with Almost Linear Architectures". In: *CoRR* abs/1902.02696 (2019). arXiv: 1902.02696.

[90] A. R. Bradley and Z. Manna. "Checking Safety by Inductive Generalization of Counterexamples to Induction". In: *Proc. of FMCAD'07*. IEEE Computer Society, 2007, pp. 173–180.

[91] A. Brüggemann-Klein, M. Murata, and D. Wood. *Regular Tree and Regular Hedge Languages over Unranked Alphabets*. Tech. rep. HKTUST-TCSC-2001-05. HKTUST, 2001.

[92] J. A. Brzozowski and E. L. Leiss. "On Equations for Regular Languages, Finite Automata, and Sequential Networks". In: *Theor. Comput. Sci.* 10 (1980), pp. 19–35.

[93] J. R. Büchi. "On a decision method in a restricted second order arithmetic". In: *Proc. of CLMPS*. Stanford Univ. Press, 1960, pp. 1–11.

[94] J. R. Büchi. "On a Decision Method in Restricted Second Order Arithmetic". In: *The Collected Works of J. Richard Büchi*. New York, NY: Springer New York, 1990, pp. 425–435.

[95] J. R. Büchi. "Weak Second-Order Arithmetic and Finite Automata". In: *Mathematical Logic Quarterly* 6.1-6 (1960), pp. 66–92. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19600060105`.

[96] J. R. Büchi. "On a Decision Method in Restricted Second-Order Arithmetic". In: *Proc. of International Congress on Logic, Methodology, and Philosophy of Science*. Stanford University Press, 1962, pp. 1–11.

[97] D. Bustan and O. Grumberg. "Simulation Based Minimization". In: *Proc. of CADE'00*. Vol. 1831. LNCS. Springer, 2000, pp. 255–270.

[98] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008), 10:1–10:38.

[99] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. "Symbolic execution for software testing in practice: preliminary assessment". In: *Proc. of ICSE'2011*. 2011, pp. 1066–1071.

[100] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. "Compositional Shape Analysis by Means of Bi-Abduction". In: *ACM Trans. Comput. Log.* 58 (2011), 26:1–26:66.

[101] A. Caulfield et al. "A Cloud-Scale Acceleration Architecture". In: *Proc. of MICRO'16*. 2016.

[102] G. Cécé. "Foundation for a series of efficient simulation algorithms". In: *Proc. of LICS'17*. IEEE Computer Society, 2017, pp. 1–12.

[103] M. Ceska, V. Havlena, L. Holík, O. Lengál, and T. Vojnar. "Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection". In: *Proc. of TACAS'18*. Vol. 10806. LNCS. Springer, 2018, pp. 155–175.

[104] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. "Alternation". In: *J. ACM* 28.1 (Jan. 1981), pp. 114–133.

[105] B. E. Chang, X. Rival, and G. C. Necula. "Shape Analysis with Structural Invariant Checkers". In: *Proc. of SAS'07*. Vol. 4634. LNCS. Springer, 2007, pp. 384–401.

[106] T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu. "What is decidable about string constraints with the ReplaceAll function". In: *PACMPL* 2.POPL (2018), 3:1–3:29.

[107] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu. "Decision procedures for path feasibility of string-manipulating programs with complex operations". In: *PACMPL* 3.POPL (2019), 49:1–49:30.

[108] Y. Chen, C. Hong, A. W. Lin, and P. Rümmer. "Learning to Prove Safety over Parameterised Concurrent Systems (Full Version)". In: *CoRR* abs/1709.07139 (2017).

[109] W. Chin, C. David, H. H. Nguyen, and S. Qin. "Automated verification of shape, size and bag properties via user-defined predicates in separation logic". In: *Sci. Comput. Program.* 77.9 (2012), pp. 1006–1036.

[110] C. R. Clark and D. E. Schimmel. "Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns". In: *FPL'03*. Springer, 2003, pp. 956–959.

[111] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 244–263.

[112] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. London, Cambridge: MIT Press, 1999.

[113] L. Clemente. "Büchi Automata Can Have Smaller Quotients". In: *Proc. of ICALP'11*. Vol. 6756. LNCS. Springer, 2011, pp. 258–270.

[114] L. Clemente and R. Mayr. "Advanced Automata Minimization". In: *Proc. of POPL'13*. ACM Trans. Comput. Log., 2013, pp. 63–74.

[115] L. Clemente and R. Mayr. "Efficient reduction of nondeterministic automata with application to language inclusion testing". In: *Logical Methods in Computer Science* 15.1 (2019).

[116] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Available on: `http://www.grappa.univ-lille3.fr/tata`. release October, 12th 2007. 2007.

[117] W. contributors. *Regular expression—Wikipedia*. 2019.

[118] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. 1st. Cambridge University Press, 2012.

[119] A. Cox and J. Leasure. "Model Checking Regular Language Constraints". In: *CoRR* abs/1708.09073 (2017).

[120] K. Culik and J. Kari. "Image compression using weighted finite automata". In: *Computers & Mathematics with Applications* 17.3 (1993), pp. 305–313.

[121] L. D'Antoni, Z. Kincaid, and F. Wang. "A Symbolic Decision Procedure for Symbolic Alternating Finite Automata". In: *CoRR* abs/1610.01722 (2016).

[122] L. D'Antoni and M. Veanes. "Static Analysis of String Encoders and Decoders". In: *Proc. of VMCAI'13*. Vol. 7737. LNCS. Springer, 2013, 209–228.

[123] D. Dams, Y. Lakhnech, and M. Steffen. "Iterating transducers". In: *J. Log. Algebr. Program.* 52-53 (2002), pp. 109–127.

[124] J. C. Davis. "Rethinking Regex Engines to Address ReDoS". In: *Proc. of ESEC/FSE'19*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, pp. 1256–1258.

[125] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee. "The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale". In: *Proc. of ESEC/FSE'18*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: ACM, 2018, pp. 246–256.

[126] L. De Moura and N. Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Commun. ACM* 54.9 (2011), pp. 69–77.

[127] J. Desharnais, F. Laviolette, and M. Tracol. "Approximate Analysis of Probabilistic Processes: Logic, Simulation and Games". In: *Proc. of QEST'08*. Sept. 2008, pp. 264–273.

[128] J. Deshmukh, E. Emerson, and P. Gupta. "Automatic Verification of Parameterized Data Structures". In: *TACAS'06*. Vol. 3920. LNCS. Springer, 2006.

[129] V. Diekert. "Makanin's Algorithm". In: *Algebraic Combinatorics on Words*. Vol. 90. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2002. Chap. 12, pp. 387–442.

[130] S. Doherty, L. Groves, V. Luchangco, and M. Moir. "Formal Verification of a Practical Lock-Free Queue Algorithm". In: *Proc. of FORTE'04*. Vol. 3235. LNCS. Springer, 2004, pp. 97–114.

[131] L. Doyen and J.-F. Raskin. "Antichain Algorithms for Finite Automata". In: *Proc. of TACAS'10*. Vol. 6015. LNCS. Springer, 2010, pp. 2–22.

[132] K. Dudka, P. Peringer, and T. Vojnar. "Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic". In: *Proc. of CAV'11*. Vol. 6806. LNCS. Springer, 2011, pp. 372–378.

[133] J. Elgaard, N. Klarlund, and A. Møller. "MONA 1.x: new techniques for WS1S and WS2S". In: *Proc. of CAV'98*. Vol. 1427. LNCS. BRICS, Department of Computer Science, Aarhus University. Springer, 1998, pp. 516–520.

[134] C. C. Elgot. "Decision Problems of Finite Automata Design and Related Arithmetics". In: *Transactions of the American Mathematical Society* 98.1 (1961), pp. 21–51.

[135] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. "Simplifying Linearizability Proofs with Reduction and Abstraction". In: *Proc. of TACAS'10*. Vol. 6015. LNCS. Springer, 2010, pp. 296–311.

[136] T. Elmas, S. Qadeer, and S. Tasiran. "A calculus of atomic actions". In: *Proc. of POPL'09*. ACM, 2009, pp. 2–15.

[137] E. A. Emerson and E. M. Clarke. "Characterizing correctness properties of parallel programs using fixpoints". In: *Automata, Languages and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 169–181.

[138] S. Exchange. *Outage Postmortem*. 2016.

[139] C. Flanagan and S. Qadeer. "Thread-Modular Model Checking". In: *Proc. of SPIN'03*. Vol. 2648. LNCS. Springer, 2003, pp. 213–224.

[140] S. Fogarty and M. Y. Vardi. "Efficient Büchi Universality Checking". In: *Proc. of TACAS'10*. Vol. 6015. LNCS. Springer, 2010, pp. 205–220.

[141] X. Fu and C. Li. "Modeling Regular Replacement for String Constraint Solving". In: *Proc. of NFM'10*. Vol. NASA/CP-2010-216215. NASA. 2010, 67–76.

[142] X. Fu, M. C. Powell, M. Bantegui, and C. Li. "Simple linear string constraints". In: *Formal Asp. Comput.* 25.6 (2013), pp. 847–891.

[143] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard. "Word Equations with Length Constraints: What's Decidable?" In: *Proc. of HVC'12*. Vol. 7857. LNCS. Springer, 2012, pp. 209–226.

[144] G. Gange, J. A. Navas, P. J. Stuckey, H. Søndergaard, and P. Schachte. "Unbounded Model-Checking with Interpolation for Regular Language Constraints". In: *Proc. of TACAS'13*. Vol. 7795. LNCS. Springer, 2013, 277–291.

[145] P. Ganty, N. Maquet, and J. Raskin. "Fixed point guided abstraction refinement for alternating automata". In: *Theor. Comput. Sci.* 411.38-39 (2010), pp. 3444–3459.

[146] P. Ganty, J. Raskin, and L. V. Begin. "A Complete Abstract Interpretation Framework for Coverability Properties of WSTS". In: *Proc. of VMCAI 2006*. Vol. 3855. LNCS. Springer, 2006, pp. 49–64.

[147] J. Glenn and W. Gasarch. "Implementing WS1S via Finite Automata". In: *Proc. of Workshop on Implementing Automata*. Vol. 1260. LNCS. Springer, 1996, pp. 50–63.

[148] GNU. *grep*. `https://www.gnu.org/software/grep/`.

[149] P. Godefroid, N. Klarlund, and K. Sen. "DART: directed automated random testing". In: *Proc. of PLDI'05*. 2005, pp. 213–223.

[150] Google. RE2. `https://github.com/google/re2`.

[151] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. "Thread-modular shape analysis". In: *Proc. of PLDI'07*. ACM, 2007, pp. 266–277.

[152] J. Graham-Cumming. *Details of the Cloudflare outage on July 2, 2019*. 2019.

[153] B. Guo, N. Vachharajani, and D. I. August. "Shape Analysis with Inductive Recursion Synthesis". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 256–265.

[154] C. Gutiérrez. "Solving Equations in Strings: On Makanin's Algorithm". In: *Proc. of LATIN*. 1998, pp. 358–373.

[155] J. Y. Halpern and V. Weissman. "Using First-Order Logic to Reason about Policies". In: *ACM Trans. Comput. Log.* 11.4 (2008), 21:1–21:41.

[156] J. Hamza, B. Jobstmann, and V. Kuncak. "Synthesis for regular specifications over unbounded domains". In: *Proc. of FMCAD'10*. IEEE Computer Society, 2010, pp. 101–109.

[157] J. Heinen, T. Noll, and S. Rieger. "Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures". In: *Electr. Notes Theor. Comput. Sci.* 266 (2010), pp. 93–107.

[158] M. Heizmann, J. Hoenicke, and A. Podelski. "Software Model Checking for People Who Love Automata". In: *Proc. of CAV'13*. Vol. 8044. LNCS. Springer, 2013, pp. 36–52.

[159] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. "Mona: Monadic Second-Order Logic in Practice". In: *Proc. of TACAS '95*. Vol. 1019. LNCS. Springer, 1995, pp. 89–110.

[160] J. G. Henriksen, M. Mukund, K. N. Kumar, and P. S. Thiagarajan. "On Message Sequence Graphs and Finitely Generated Regular MSC Languages". In: *Proc. of ICALP'00*. Vol. 1853. LNCS. Springer, 2000, pp. 675–686.

[161] J. G. Henriksen, M. Mukund, K. N. Kumar, and P. S. Thiagarajan. "Regular Collections of Message Sequence Charts". In: *Proc. of MFCS'00*. Vol. 1893. LNCS. Springer, 2000, pp. 405–414.

[162] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. "Computing Simulations on Finite and Infinite Graphs". In: *Proc. of FOCS'95*. Washington, DC, USA: IEEE, 1995, pp. 453–462.

[163] K. Hoder and N. Bjørner. "Generalized Property Directed Reachability". In: *Proc. of SAT'12*. Vol. 7317. LNCS. Springer, 2012, pp. 157–171.

[164] J. Högberg, A. Maletti, and J. May. "Backward and Forward Bisimulation Minimisation of Tree Automata". In: *Proc. of CIAA'07*. Vol. 4783. LNCS. Springer, 2007, pp. 109–121.

[165] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. "Fast and Precise Sanitizer Analysis with BEK". In: *Proc. of USENIX'11*. USENIX Association, 2011.

[166] P. Hooimeijer and W. Weimer. "StrSolve: Solving string constraints lazily". In: *Autom. Softw. Eng.* 19.4 (2012), pp. 531–559.

[167] H. Hosoya, J. Vouillon, and B. C. Pierce. "Regular expression types for XML". In: *ACM Trans. Program. Lang. Syst.* 27.1 (2005), pp. 46–90.

[168] D. Hovland. "Regular Expressions with Numerical Constraints and Automata with Counters". In: *Proc. of ICTAC*. Vol. 5684. LNCS. Springer, 2009, pp. 231–245.

[169] T. Hune and A. Sandholm. "A Case Study on Using Automata in Control Synthesis". In: *Proc. of FASE'00*. Vol. 1783. LNCS. Springer, 2000, 349–362.

[170] "IEEE Standard for Ethernet - Amendment 10: Media Access Control Parameters, Physical Layers, and Management Parameters for 200 Gb/s and 400 Gb/s Operation". In: *IEEE Std 802.3bs-2017* (2017), pp. 1–372.

[171] L. Ilie and S. Yu. "Follow automata". In: *Information and Computation* 186.1 (2003), pp. 146–162.

[172] R. Iosif, A. Rogalewicz, and J. Šimáček. "The Tree Width of Separation Logic with Recursive Definitions". In: *Proc. of CADE'13*. Vol. 7898. LNCS. Springer, 2013, pp. 21–38.

[173] R. Iosif, A. Rogalewicz, and T. Vojnar. "Deciding Entailments in Inductive Separation Logic with Tree Automata". In: *Proc. of ATVA'14*. Vol. 8837. LNCS. Springer, 2014, pp. 201–218.

[174] A. Jez. "Recompression: A Simple and Powerful Technique for Word Equations". In: *J. ACM* 63.1 (2016), 4:1–4:51.

[175] C. B. Jones. "Tentative Steps Toward a Development Method for Interfering Programs". In: *ACM Trans. Program. Lang. Syst.* 5.4 (1983), pp. 596–619.

[176] B. Jonsson. "Using refinement calculus techniques to prove linearizability". In: *Formal Asp. Comput.* 24.4-6 (2012), pp. 537–554.

[177] B. Jonsson and M. Nilsson. "Transitive Closures of Regular Relations for Verifying Infinite-State Systems". In: *Proc. of TACAS'00*. Vol. 1785. LNCS. Springer, 2000, pp. 220–234.

[178] A. Kaiser, D. Kroening, and T. Wahl. "Efficient Coverability Analysis by Proof Minimization". In: *Proc. of CONCUR 2012*. Vol. 7454. LNCS. Springer, 2012, pp. 500–515.

[179] J. Katelaan, C. Matheja, and F. Zuleger. "Effective Entailment Checking for Separation Logic with Inductive Definitions". In: *Proceedings of TACAS'19*. Vol. 11428. LNCS. Springer, 2019, pp. 319–336.

[180] S. Kausler and E. Sherman. "Evaluation of String Constraint Solvers in the Context of Symbolic Execution". In: *Proc. of ASE'14*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 259–270.

[181] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. "Symbolic model checking with rich assertional languages". In: *Theor. Comput. Sci.* 256.1-2 (2001), pp. 93–112.

[182] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. "HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars". In: *ACM Trans. Comput. Log.* 21.4 (2012), 25:1–25:28.

[183] P. Kilpeläinen and R. Tuhkanen. "One-unambiguity of regular expressions with numeric occurrence indicators". In: *Information and Computation* 205.6 (2007), pp. 890–916.

[184] N. Klarlund, M. Nielsen, and K. Sunesen. "A Case Study in Automated Verification Based on Trace Abstractions". In: *Formal System Specification, The RPC-Memory Specification Case Study*. Vol. 1169. LNCS. Springer Verlag, 1996.

[185] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3. BRICS, Department of Computer Science, Aarhus University. Jan. 2001.

[186] J. Kloos, R. Majumdar, F. Niksic, and R. Piskac. "Incremental, Inductive Coverability". In: *Proc. of CAV 2013*. Vol. 8044. LNCS. Springer, 2013, 158–173.

[187] D. Kozen. *Automata and computability*. UTiCS. Springer, 1997.

[188] D. Kuske. "A Further Step towards a Theory of Regular MSC Languages". In: *Proc. of STACS'02*. Vol. 2285. LNCS. Springer, 2002, pp. 489–500.

[189] Q. L. Le, C. Gherghina, S. Qin, and W. Chin. "Shape Analysis via Second-Order Bi-Abduction". In: *Proc. of CAV'14*. Vol. 8559. LNCS. Springer, 2014, pp. 52–68.

[190] O. Lee, H. Yang, and R. Petersen. "Program Analysis for Overlaid Data Structures". In: *Proc. of CAV'11*. Vol. 6806. LNCS. Springer, 2011, pp. 592–608.

[191] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. "A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions". In: *Proc. of CAV'14*. Vol. 8559. LNCS. Springer, 2014, pp. 646–662.

[192] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, and M. Deters. "An efficient SMT solver for string constraints". In: *Formal Methods in System Design* 48.3 (2016), pp. 206–234.

[193] T. Liang, N. Tsiskaridze, A. Reynolds, C. Tinelli, and C. Barrett. "A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings". In: *Proc. of FroCoS'15*. Vol. 9322. LNCS. Springer, 2015, pp. 135–150.

[194] A. W. Lin and P. Barceló. "String solving with word equations and transducers: Towards a logic for analysing mutation XSS". In: *Proc. of POPL'16*. ACM Trans. Comput. Log., 2016, pp. 123–136.

[195] C. Lin, C. Huang, C. Jiang, and S. Chang. "Optimization of Pattern Matching Circuits for Regular Expression on FPGA". In: *IEEE Trans. VLSI Syst.* 15.12 (2007), pp. 1303–1310.

[196] A. Loginov, T. Reps, and M. Sagiv. "Abstraction Refinement via Inductive Learning". In: *Proc. of CAV'05*. Vol. 3576. LNCS. Springer, 2005, pp. 519–533.

[197] B. Loring, D. Mitchell, and J. Kinder. "ExpoSE: Practical Symbolic Execution of Standalone JavaScript". In: *Proc. of SPIN*. 2017.

[198] M. Roesch et al. *Snort: A Network Intrusion Detection and Prevention System*. `http://www.snort.org`. Cisco and/or its affiliates.

[199] P. Madhusudan, G. Parlato, and X. Qiu. "Decidable Logics Combining Heap Structures and Data". In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 611–622.

[200] P. Madhusudan, G. Parlato, and X. Qiu. "Decidable logics combining heap structures and data". In: *Proc. of POPL'11*. ACM Trans. Comput. Log., 2011, pp. 611–622.

[201] P. Madhusudan and X. Qiu. "Efficient Decision Procedures for Heaps Using STRAND". In: *Proc. of SAS'11*. Vol. 6887. LNCS. Springer, 2011, pp. 43–59.

[202] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. "Automatic Numeric Abstractions for Heap-manipulating Programs". In: *Proc. of POPL'10*. ACM, 2010, pp. 211–222.

[203] G. S. Makanin. "The problem of solvability of equations in a free semigroup". In: *Sbornik: Mathematics* 32.2 (1977), pp. 129–198.

[204] V. Malík, M. Hruska, P. Schrammel, and T. Vojnar. "Template-Based Verification of Heap-Manipulating Programs". In: *Proc. of FMCAD'18*. IEEE, 2018, pp. 1–9.

[205] A. Malkis, A. Podelski, and A. Rybalchenko. "Thread-Modular Verification Is Cartesian Abstract Interpretation". In: *Proc. of ICTAC'06*. Vol. 4281. LNCS. Springer, 2006, pp. 183–197.

[206] C. Matheja, C. Jansen, and T. Noll. "Tree-Like Grammars and Separation Logic". In: *Proc. of APLAS'15*. Vol. 9458. LNCS. Springer, 2015, pp. 90–108.

[207] D. Matousek, J. Korenek, and V. Pus. "High-speed Regular Expression Matching with Pipelined Automata". In: *Proc. of FPT'16*. IEEE, 2016, pp. 93–100.

[208] Matt Jonkman et al. Suricata. Emerging Threats, 2017.

[209] R. Mayr and L. Clemente. "Advanced automata minimization". In: *Proc. of POPL 2013*. ACM, 2013, pp. 63–74.

[210] R. Meyer and S. Wolff. "Pointer life cycle types for lock-free data structures with memory reclamation". In: *PACMPL* 4.POPL (2020), 68:1–68:36.

[211] M. M. Michael and M. L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: *Proc. of PODC'96*. ACM, 1996, pp. 267–275.

[212] A. Møller and M. I. Schwartzbach. "The Design Space of Type Checkers for XML Transformation Languages". In: *Proc. of ICDT'05*. 2005, pp. 17–36.

[213] A. Møller and M. I. Schwartzbach. "The Pointer Assertion Logic Engine". In: *Proc. of PLDI'01*. ACM Trans. Comput. Log., 2001, pp. 221–231.

[214] F. Morawietz and T. Cornell. "The Logic-Automaton Connection in Linguistics". In: *Proc. of LACL'97*. Vol. 1582. LNAI. Springer Verlag, 1997.

[215] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. "Automated Verification of Shape and Size Properties Via Separation Logic". In: *VMCAI'07*. Vol. 4349. LNCS. Springer, 2007, pp. 251–266.

[216] OWASP. `https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf`. 2013.

[217] R. Paige and R. E. Tarjan. "Three Partition Refinement Algorithms". In: *SIAM J. Comput.* 16.6 (1987), pp. 973–989.

[218] W. Plandowski. "An efficient algorithm for solving word equations". In: *Proc. of STOC*. 2006, pp. 467–476.

[219] W. Plandowski. "Satisfiability of word equations with constants is in PSPACE". In: *J. ACM* 51.3 (2004), pp. 483–496.

[220] A. Pnueli. "The Temporal Logic of Programs". In: *Proc. of FOCS'77*. IEEE Computer Society, 1977, pp. 46–57.

[221] A. Pnueli and R. Rosner. "On the Synthesis of a Reactive Module". In: *Proc. of POPL'89*. ACM Press, 1989, pp. 179–190.

[222] A. Podelski and T. Wies. "Counterexample-Guided Focus". In: *Proc. of POPL'10*. ACM, 2010, pp. 249–260.

[223] C. Popeea, A. Rybalchenko, and A. Wilhelm. "Reduction for compositional verification of multi-threaded programs". In: *Proc. of FMCAD'14*. IEEE, 2014, pp. 187–194.

[224] A. Putnam et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *Proc. of ISCA'14*. IEEE Press, 2014, pp. 13–24.

[225] J. P. Queille and J. Sifakis. "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 337–351.

[226] M. O. Rabin. "Decidability of second-order theories and automata on infinite trees". In: *Bull. Amer. Math. Soc.* 74.5 (Sept. 1968), pp. 1025–1029.

[227] M. O. Rabin. *Automata on Infinite Objects and Church's Problem*. USA: American Mathematical Society, 1972.

[228] F. Ranzato and F. Tapparo. "An efficient simulation algorithm based on abstract interpretation". In: *Information and Computation* 208 (2010), pp. 1–22.

[229] G. Redelinghuys, W. Visser, and J. Geldenhuys. "Symbolic execution of programs with strings". In: *Proc. of SAICSIT*. 2012, pp. 139–148.

[230] RegExLib.com. *The Internet's first Regular Expression Library*, `http://regexlib.com/`.

[231] J. C. Reynolds. "Separation logic: a logic for shared mutable data structures". In: *Proc. of LICS'02*. July 2002, pp. 55–74.

[232] Robin Sommer et al. *The Bro Network Security Monitor*. `http://www.bro.org`.

[233] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. "TaDA: A Logic for Time and Data Abstraction". In: *Proc. of ECOOP'14*. Vol. 8586. LNCS. Springer, 2014, pp. 207–231.

[234] M. Sagiv, T. Reps, and R. Wilhelm. "Parametric Shape Analysis via 3-valued Logic". In: *ACM Transactions on Programming Languages and Systems* 24.3 (2002), pp. 217–298.

[235] A. Sandholm and M. I. Schwartzbach. "Distributed Safety Controllers for Web Services". In: *Proc. of FASE'98*. Springer, 1998, pp. 270–284.

[236] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. "A Symbolic Execution Framework for JavaScript". In: *Proc. of S&P'10*. IEEE Computer Society, 2010, pp. 513–528.

[237] G. Schellhorn, J. Derrick, and H. Wehrheim. "A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures". In: *ACM Trans. Comput. Log.* 15.4 (2014), 31:1–31:37.

[238] D. Seese. "The Structure of Models of Decidable Monadic Theories of Graphs". In: *Ann. Pure Appl. Logic* 53.2 (1991), pp. 169–195.

[239] M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. "Abstract Transformers for Thread Correlation Analysis". In: *Proc. of APLAS'09*. Vol. 5904. LNCS. Springer, 2009, pp. 30–46.

[240] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs. "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript". In: *Proceedings of ESEC/FSE'13*. 2013, pp. 488–498.

[241] R. Sidhu and V. K. Prasanna. "Fast Regular Expression Matching Using FPGAs". In: *Proc. of FCCM'01*. IEEE Computer Society, 2001, pp. 227–238.

[242] M. Sipser. *Introduction to Theory of Computation*. Vol. 2. Thomson Course Technology Boston, 2006.

[243] A. P. Sistla, M. Y. Vardi, and P. Wolper. "The Complementation Problem for Büchi Automata with Applications to Temporal Logic (Extended Abstract)". In: *Proc. of ICALP'85*. Vol. 194. LNCS. Springer, 1985, pp. 465–474.

[244] M. A. Smith and N. Klarlund. "Verification of a Sliding Window Protocol Using IOA and MONA". In: *Proc. of FORTE/PSTV'00*. Vol. 183. IFIP. Kluwer, 2000, pp. 19–34.

[245] R. Smith, C. Estan, S. Jha, and I. Siahaan. "Fast Signature Matching Using Extended Finite Automaton (XFA)". In: *Proc. of ICISS'08*. Vol. 5352. LNCS. Springer, 2008, pp. 158–172.

[246] I. Sourdis, J. Bispo, J. M. P. Cardoso, and S. Vassiliadis. "Regular Expression Matching in Reconfigurable Hardware". In: *Journal of Signal Processing Systems* 51.1 (2008), pp. 99–121.

[247] H. Spencer. "Software Solutions in C". In: San Diego, CA, USA: Academic Press Professional, Inc., 1994. Chap. A Regular-expression Matcher, 35–71.

[248] M. Sperberg-McQueen. *Notes on finite state automata with counters*. `https://www.w3.org/XML/2004/05/msm-cfa.html`. Accessed: 2018-08-08.

[249] T. Strazny. "An algorithmic framework for checking coverability in well-structured transition systems". PhD thesis. Universität Oldenburg, 2014.

[250] H. Subramanian and P. Shankar. "Compressing XML Documents Using Recursive Finite State Automata". In: *Proc. of CIAA'05*. 2005, pp. 282–293.

[251] M. Takahashi. "Generalizations of Regular Sets and Their Application to a Study of Context-Free Languages". In: *Inf. Comput.* 27.1 (1975), pp. 1–36.

[252] T. Tateishi, M. Pistoia, and O. Tripp. "Path- and index-sensitive string analysis based on monadic second-order logic". In: *ACM Trans. Comput. Log.* 22.4 (2013), 33:1–33:33.

[253] J. W. Thatcher. "Characterizing Derivation Trees of Context-Free Grammars through a Generalization of Finite Automata Theory". In: *J. Comput. Syst. Sci.* 1.4 (1967), pp. 317–322.

[254] J. W. Thatcher and J. B. Wright. "Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic". In: *Mathematical Systems Theory* 2.1 (1968), pp. 57–81.

[255] W. Thomas. "Languages, Automata, and Logic". In: *Handbook of Formal Languages: Volume 3 Beyond Words*. Springer, 1997, pp. 389–455.

[256] K. Thompson. "Programming Techniques: Regular Expression Search Algorithm". In: *Commun. ACM* 11.6 (June 1968), pp. 419–422.

[257] T. Touili. "Regular Model Checking using Widening Techniques". In: *Electr. Notes Theor. Comput. Sci.* 50.4 (2001), pp. 342–356.

[258] R. K. Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research ..., 1986.

[259] M. Trinh, D. Chu, and J. Jaffar. "Progressive Reasoning over Recursively-Defined Strings". In: *Proc. of CAV'16*. Vol. 9779. LNCS. Springer, 2016, pp. 218–240.

[260] M. Trinh, D. Chu, and J. Jaffar. "S3: A Symbolic String Solver for Vulnerability Detection in Web Applications". In: *Proc. of CCS*. ACM Trans. Comput. Log., 2014, pp. 1232–1243.

[261] V. Vafeiadis. "RGSep Action Inference". In: *Proc. of VMCAI'10*. Vol. 5944. LNCS. Springer, 2010.

[262] V. Vafeiadis. "RGSep Action Inference". In: *Proc. of VMCAI'10*. Vol. 5944. LNCS. Springer, 2010, pp. 345–361.

[263] V. Vafeiadis. "Shape-Value Abstraction for Verifying Linearizability". In: *Proc. of VMCAI'09*. Vol. 5403. LNCS. Springer, 2009, pp. 335–348.

[264] V. Vafeiadis and M. J. Parkinson. "A Marriage of Rely/Guarantee and Separation Logic". In: *Proc. of CONCUR '07*. Vol. 4703. LNCS. Springer, 2007, pp. 256–271.

[265] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. "The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware". In: *Proc. of RAID'07*. Springer, 2007, pp. 107–126.

[266] A. Valmari. "Simple Bisimilarity Minimization in $\mathcal{O}(m \log n)$ Time". In: *Fundam. In shape.* 105.3 (2010), pp. 319–339.

[267] M. Y. Vardi. "An automata-theoretic approach to linear temporal logic". In: *Logics for Concurrency: Structure versus Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 238–266.

[268] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. "Symbolic finite state transducers: Algorithms and applications". In: *Proc. of POPL'12*. ACM Trans. Comput. Log., 2012, pp. 137–150.

[269] M. Veanes, O. Saarikivi, E. Xu, and T. Wan. "Symbolic Regex Matcher". In: *Proc. of TACAS*. 2019.

[270] H. Wang, T. Tsai, C. Lin, F. Yu, and J. R. Jiang. "String Analysis via Automata Manipulation with Logic Circuit Representation". In: *Proc. of CAV'16*. Vol. 9779. LNCS. Springer, 2016, pp. 241–260.

[271] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. "Dynamic test input generation for web applications". In: *Proc. of ISSTA'08*. ACM Trans. Comput. Log., 2008, pp. 249–260.

[272] A. D. Weinert. "Inferring Heap Abstraction Grammars". BSc thesis. RWTH Aachen, 2012.

[273] T. Wies, M. Muñiz, and V. Kuncak. "An Efficient Decision Procedure for Imperative Tree Data Structures". In: *Proc. of CADE'11*. Vol. 6803. LNCS. Springer, 2011, pp. 476–491.

[274] P. Wolper. "Expressing Interesting Properties of Programs in Propositional Temporal Logic". In: *Proc. of POPL'86*. ACM Press, 1986, pp. 184–193.

[275] P. Wolper. "On the Use of Automata for Deciding Linear Arithmetic". In: *Proc. of TABLEAUX'09*. Vol. 5607. LNCS. Springer, 2009, p. 16.

[276] M. D. Wulf, L. Doyen, and J.-F. Raskin. "A Lattice Theory for Solving Games of Imperfect Information". In: *Proc. of HSCC'06*. Vol. 3927. LNCS. Springer Verlag, 2006.

[277] M. D. Wulf, L. Doyen, T. A. Henzinger, and J. Raskin. "Antichains: A New Algorithm for Checking Universality of Finite Automata". In: *Proc. of CAV'06*. Vol. 4144. LNCS. Springer, 2006, pp. 17–30.

[278] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. "Scalable Shape Analysis for Systems Code". In: *Proc. of CAV'08*. Vol. 5123. LNCS. Springer, 2008, pp. 385–398.

[279] F. Yu, M. Alkhalaf, and T. Bultan. "Stranger: An Automata-Based String Analysis Tool for PHP". In: *Proc. of TACAS'10*. Vol. 6015. LNCS. Springer, 2010, pp. 154–157.

[280] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. "Automata-based symbolic string analysis for vulnerability detection". In: *Formal Methods in System Design* 44.1 (2014), pp. 44–70.

[281] F. Yu, T. Bultan, and O. H. Ibarra. "Relational String Verification Using Multi-Track Automata". In: *Int. J. Found. Comput. Sci.* 22.8 (2011), pp. 1909–1924.

[282] F. Yu, T. Bultan, and O. H. Ibarra. "Symbolic String Verification: Combining String Analysis and Size Analysis". In: *Proc. of TACAS'09*. Vol. 5505. LNCS. Springer, 2009, pp. 322–336.

[283] S. Yun and K. Lee. "Optimization of Regular Expression Pattern Matching Circuit Using At-Most Two-Hot Encoding on FPGA". In: *Proc. of FPL'10*. 2010.

[284] K. Zee, V. Kuncak, and M. C. Rinard. "Full functional verification of linked data structures". In: *Proc. of PLDI'08*. ACM, 2008, pp. 349–361.

[285] S. J. Zhang and Y. Liu. "Model Checking a Lazy Concurrent List-Based Set Algorithm". In: *Proc. of SSIRI'10*. IEEE Computer Society, 2010, pp. 43–52.

[286] Y. Zheng, X. Zhang, and V. Ganesh. "Z3-str: A Z3-based string solver for web application analysis". In: *Proc. of ESEC/FSE'13*. ACM Trans. Comput. Log., 2013, pp. 114–124.

[287] M. Zhou, F. He, B. Wang, M. Gu, and J. Sun. "Array Theory of Bounded Elements and its Applications". In: *J. Autom. Reasoning* 52.4 (2014), 379–405.

Appendix A

Selected Papers

For each section of this thesis discussing a particular research direction, we have selected one or two representative papers to be attached to the thesis. The selection is based on the overall importance in the repsective line of research. In the case of Sections 2.2, 2.3, and 2.4, the selection was adjusted in order to avoid an intersection with the author's dissertation [9]. The following papers were selected:

[8] R. Almeida, L. Holík, and R. Mayr. "Reduction of Nondeterministic Tree Automata". In: *Proc. of TACAS'16*. Vol. 9636. LNCS. Springer, 2016, 717–735.

[15] L. Holík, O. Lengál, J. Šimáček, and T. Vojnar. "Efficient Inclusion Checking on Explicit and Semi-symbolic Tree Automata". In: *Proc. of ATVA'11*. Vol. 6996. LNCS. Springer Berlin Heidelberg, 2011, pp. 243–258.

[18] L. Holík, P. Janků, A. W. Lin, P. Rümmer, and T. Vojnar. "String constraints with concatenation and transducers solved efficiently". In: *PACMPL* 2.POPL (2018), 4:1–4:32.

[20] L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar. *Regex Matching with Counting-Set Automata*. accepted at OOPSLA'20. 2020.

[29] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. "Forest Automata for Verification of Heap Manipulation". In: *Formal Methods in System Design* 1 (2012), pp. 83–106.

[40] P. Abdulla, M. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. "String Constraints for Verification". English. In: *Proc. of CAV'14*. Vol. 8559. LNCS. Springer International Publishing, 2014, pp. 150–166.

[42] P. A. Abdulla, F. Haziza, and L. Holík. "All for the Price of Few". In: *Proc. of VMCAI'13*. Vol. 7737. LNCS. Springer Berlin Heidelberg, 2013, pp. 476–495.

[46] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. "An Integrated Specification and Verification Technique for Highly Concurrent Data Structures". In: *Int. J. Softw. Tools Technol. Transf.* 19.5 (2017), pp. 549–563.

Reduction of Nondeterministic Tree Automata

Ricardo Almeida[1], Lukáš Holík[2], and Richard Mayr[1( )]

[1] University of Edinburgh, Edinburgh, UK
rmayr@staffmail.ed.ac.uk
[2] Brno University of Technology, Brno, Czech Republic

Abstract. We present an efficient algorithm to reduce the size of nondeterministic tree automata, while retaining their language. It is based on new transition pruning techniques, and quotienting of the state space w.r.t. suitable equivalences. It uses criteria based on combinations of downward and upward simulation preorder on trees, and the more general downward and upward language inclusions. Since tree-language inclusion is EXPTIME-complete, we describe methods to compute good approximations in polynomial time.

We implemented our algorithm as a module of the well-known libvata tree automata library, and tested its performance on a given collection of tree automata from various applications of libvata in regular model checking and shape analysis, as well as on various classes of randomly generated tree automata. Our algorithm yields substantially smaller and sparser automata than all previously known reduction techniques, and it is still fast enough to handle large instances.

1 Introduction

Background. Tree automata are a generalization of word automata that accept trees instead of words [14]. They have many applications in model checking [5,6,12], term rewriting [15], and related areas of formal software verification, e.g., shape analysis [3,18,20]. Several software packages for manipulating tree automata have been developed, e.g., MONA [9], Timbuk [16], Autowrite [15] and libvata [22], on which other verification tools like Forester [23] are based.

For nondeterministic automata, many questions about their languages are computationally hard. The language universality, equivalence and inclusion problems are PSPACE-complete for word automata and EXPTIME-complete for tree automata [14]. However, recently techniques have been developed that can solve many practical instances fairly efficiently. For word automata there are antichain techniques [2], congruence-based techniques [10] and techniques based on generalized simulation preorders [13]. The antichain techniques have been generalized to tree automata in [11,21] and implemented in the libvata library [22]. Performance problems also arise in computing the intersection of several languages, since the product construction multiplies the numbers of states.

This work was supported by the Czech Science Foundation, project 16-24707Y.

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 717–735, 2016.
DOI: 10.1007/978-3-662-49674-9_46

Automata Reduction. Our goal is to make tree automata more computationally tractable in practice. We present an efficient algorithm for the reduction of nondeterministic tree automata, in the sense of obtaining a smaller automaton with the same language, though not necessarily with the absolute minimal possible number of states. (In general, there is no unique nondeterministic automaton with the minimal possible number of states for a given language, i.e., there can be several non-isomorphic nondeterministic automata of minimal size. This holds even for word automata.) The reason to perform reduction is that the smaller reduced automaton is more efficient to handle in a subsequent computation. Thus there is an algorithmic tradeoff between the effort for reduction and the complexity of the problem later considered for this automaton. The main applications of reduction are the following: (1) Helping to solve hard problems like language universality/equivalence/inclusion. (2) If automata undergo a long chain of manipulations/combinations by operations like union, intersection, projection, etc., then intermediate results can be reduced several times on the way to keep the automata within a manageable size. (3) There are fixed-parameter tractable problems (e.g., in model checking where an automaton encodes a logic formula) where the size of one automaton very strongly influences the overall complexity, and must be kept as small as possible.

Our Contribution. We present a reduction algorithm for nondeterministic tree automata. (The tool is available for download [7].) It is based on a combination of new transition pruning techniques for tree automata, and quotienting of the state space w.r.t. suitable equivalences. The pruning techniques are related to those presented for word automata in [13], but significantly more complex due to the fundamental asymmetry between the upward and downward directions in trees.

Transition pruning in word automata [13] is based on the observation that certain transitions can be removed (a.k.a pruned) without changing the language, because other 'better' transitions remain. One defines some strict partial order (p.o.) between transitions and removes all transitions that are not maximal w.r.t. this order. A strict p.o. between transitions is called *good for pruning* (GFP) iff pruning w.r.t. it preserves the language of the automaton. Note that pruning reduces not only the number of transitions, but also, indirectly, the number of states. By removing transitions, some states may become 'useless', in the sense that they are unreachable from any initial state, or that it is impossible to reach any accepting state from them. Such useless states can then be removed from the automaton without changing its language. One can obtain computable strict p.o. between transitions by comparing the possible backward- and forward behavior of their source- and target states, respectively. For this, one uses computable relations like backward/forward simulation preorder and approximations of backward/forward trace inclusion via lookahead- or multipebble simulations. Some such combinations of backward/forward trace/simulation orders on states induce strict p.o. between transitions that are GFP, while others do not [13].

However, there is always a symmetry between backward and forward, since finite words can equally well be read in either direction.

This symmetry does not hold for tree automata, because the tree branches as one goes downward, while it might 'join in' side branches as one goes upward. While downward simulation preorder (resp. downward language inclusion) between states in a tree automaton is a direct generalization of forward simulation preorder (resp. forward language inclusion) on words, the corresponding upward notions do not correspond to backward on words. Comparing upward behavior of states in tree automata depends also on the branches that 'join in' from the sides as one goes upward in the tree. Thus upward simulation/language inclusion is only defined *relative* to a given other relation that compares the downward behavior of states 'joining in' from the sides [1]. So one speaks of "upward simulation *of* the identity relation" or "upward simulation *of* downward simulation". When one studies strict p.o. between transitions in tree automata in order to check whether they are GFP, one has combinations of three relations: the source states are compared by an upward relation $X(Y)$ of some downward relation Y, while the target states are compared w.r.t. some downward relation Z (where Z can be, and often must be, different from Y). This yields a richer landscape, and many counter-intuitive effects.

We provide a complete picture of which combinations of upward/downward simulation/trace inclusions are GFP on tree automata; cf. Fig. 4. Since tree-(trace)language inclusion is `EXPTIME`-complete [14], we describe methods to compute good approximations of them in polynomial time. Finally, we also generalize results on quotienting of tree automata [19] to larger relations, such as approximations of trace inclusion.

We implemented our algorithm [7] as a module of the well-known `libvata` [22] tree automaton library, and tested its performance on a given collection of tree automata from various applications of `libvata` in regular model checking and shape analysis, as well as on various classes of randomly generated tree automata. Our algorithm yields substantially smaller automata than all previously known reduction techniques (which are mainly based on quotienting). Moreover, the thus obtained automata are also much sparser (i.e., use fewer transitions per state and less nondeterministic branching) than the originals, which yields additional performance advantages in subsequent computations.

2 Trees and Tree Automata

Trees. A *ranked alphabet* Σ is a set of symbols together with a function $\# : \Sigma \to \mathbb{N}_0$. For $a \in \Sigma$, $\#(a)$ is called the *rank* of a. For $n \geq 0$, we denote by Σ_n the set of all symbols of Σ which have rank n.

We define a *node* as a sequence of elements of $\mathbb{N}$, where ε is the empty sequence. For a node $v \in \mathbb{N}^*$, any node v' s.t. $v = v'v''$, for some node v'', is said to be a *prefix* of v, and if $v'' \neq \varepsilon$ then v' is a *strict prefix* of v. For a node $v \in \mathbb{N}^*$, we define the i-th child of v to be the node vi, for some $i \in \mathbb{N}$. Given a ranked alphabet Σ, a *tree* over Σ is defined as a partial mapping $t : \mathbb{N}^* \to \Sigma$

such that for all $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, if $vi \in dom(t)$ then **(1)** $v \in dom(t)$, and **(2)** $\#(t(v)) \geq i$. In this paper we consider only finite trees.

Note that the number of children of a node v may be smaller than $\#(t(v))$. In this case we say that the node is *open*. Nodes which have exactly $\#(t(v))$ children are called *closed*. Nodes which do not have any children are called *leaves*. A tree is closed if all its nodes are closed, otherwise it is open. By $\mathbb{C}(\Sigma)$ we denote the set of all closed trees over Σ and by $\mathbb{T}(\Sigma)$ the set of all trees over Σ. A tree t is *linear* iff every node in $dom(t)$ has at most one child.

The *subtree* of a tree t at v is defined as the tree t_v such that $dom(t_v) = \{v' \mid vv' \in dom(t)\}$ and $t_v(v') = t(vv')$ for all $v' \in dom(t_v)$. A tree t' is a prefix of t iff $dom(t') \subseteq dom(t)$ and for all $v \in dom(t')$, $t'(v) = t(v)$. For $t \in \mathbb{C}(\Sigma)$, the *height of a node* v of t is given by the function h: if v is a leaf then $h(v) = 1$, otherwise $h(v) = 1 + max(h(v1)), \ldots, h(v\#(t(v))))$. We define the height of a tree $t \in \mathbb{C}(\Sigma)$ as $h(\epsilon)$, i.e., as the number of levels of t.

Tree Automata, Top-Down. A (finite, nondeterministic) *top-down tree automaton* (TDTA) is a quadruple $A = (\Sigma, Q, \delta, I)$ where Q is a finite set of states, $I \subseteq Q$ is a set of initial states, Σ is a ranked alphabet, and $\delta \subseteq Q \times \Sigma \times Q^+$ is the set of transition rules. A TDTA has an unique final state, which we represent by ψ. The transition rules satisfy that if $\langle q, a, \psi \rangle \in \delta$ then $\#(a) = 0$, and if $\langle q, a, q_1 \ldots q_n \rangle \in \delta$ (with $n > 0$) then $\#(a) = n$.

A *run* of A over a tree $t \in \mathbb{T}(\Sigma)$ (or a t-run in A) is a partial mapping $\pi : \mathbb{N}^* \to Q$ such that $v \in dom(\pi)$ iff either $v \in dom(t)$ or $v = v'i$ where $v' \in dom(t)$ and $i \leq \#(t(v'))$. Further, for every $v \in dom(t)$, there exists either **(a)** a rule $\langle q, a, \psi \rangle$ such that $q = \pi(v)$ and $a = t(v)$, or **(b)** a rule $\langle q, a, q_1 \ldots q_n \rangle$ such that $q = \pi(v)$, $a = t(v)$, and $q_i = \pi(vi)$ for each $i : 1 \leq i \leq \#(a)$. A *leaf of a run* π on t is a node $v \in dom(\pi)$ such that $vi \in dom(\pi)$ for no $i \in \mathbb{N}$. We call it *dangling* if $v \notin dom(t)$. Intuitively, the dangling nodes of a run over t are all the nodes which are in π but are missing in t due to it being incomplete. Notice that dangling leaves of π are children of open nodes of t. The prefix of depth k of a run π is denoted π_k. Runs are always finite since the trees we are considering are finite.

We write $t \overset{\pi}{\Longrightarrow} q$ to denote that π is a t-run of A such that $\pi(\epsilon) = q$. We use $t \Longrightarrow q$ to denote that such run π exists. A run π is accepting if $t \overset{\pi}{\Longrightarrow} q \in I$. The *downward language of a state* q in A is defined by $D_A(q) = \{t \in \mathbb{C}(\Sigma) \mid t \Longrightarrow q\}$, while the *language* of A is defined by $L(A) = \bigcup_{q \in I} D_A(q)$. The *upward language* of a state q in A, denoted $U_A(q)$, is then defined as the set of open trees t, such that there exists an accepting t-run π with exactly one dangling leaf v s.t. $\pi(v) = q$. We omit the A subscript notation when it is implicit which automaton we are considering.

In the related literature, it is common to define a tree automaton bottom-up, reading a tree from the leaves to the root [11,14,21]. A bottom-up tree automaton (BUTA) can be obtained from a TDTA by reversing the direction of the transition rules and by swapping the roles between the initial states and the final states. See [8] for an example of a tree automaton presented in both BUTA and TDTA form.

3 Simulations and Trace Inclusions

We consider different types of relations on states of a TDTA which underapproximate language inclusion. Note that words are but a special case of trees where every node has only one child, i.e., words are linear trees. *Downward* simulation/trace inclusion on TDTA corresponds to *direct forward* simulation/trace inclusion in special case of word automata, and *upward* corresponds to *backward* [13].

Forward Simulation on Word Automata. Let $A = (\Sigma, Q, \delta, I, F)$ be a NFA. A *direct forward simulation* D is a binary relation on Q such that if $q\ D\ r$, then

1. $q \in F \implies r \in F$, and
2. for any $\langle q, a, q' \rangle \in \delta$, there exists $\langle r, a, r' \rangle \in \delta$ such that $q'\ D\ r'$.

The set of direct forward simulations on A contains *id* and is closed under union and transitive closure. Thus there is a unique maximal direct forward simulation on A, which is a preorder. We call it *the direct forward simulation preorder on* A and write $\sqsubseteq^{\mathsf{di}}$.

Forward Trace Inclusion on Word Automata. Let $A = (\Sigma, Q, \delta, I, F)$ be a NFA and $w = \sigma_1\, \sigma_2 \ldots \sigma_n \in \Sigma^*$ a word of length n. A trace of A on w (or a w-trace) starting at q is a sequence of transitions $\pi = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q_n$ such that $q_0 = q$. The *direct forward trace inclusion* preorder $\subseteq^{di}$ is a binary relation on Q such that $q \subseteq^{di} r$ iff

1. $(q \in F \implies r \in F)$, and
2. for every word $w = \sigma_1\, \sigma_2 \ldots \sigma_n \in \Sigma^*$ and for every w-trace (starting at q) $\pi_q = q \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q_n$, there exists a w-trace (starting at r) $\pi_r = r \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} r_n$ such that $(q_i \in F \implies r_i \in F)$ for each $i : 1 \leq i \leq n$.

Since π_r is required to preserve the acceptance of the states in π_q, trace inclusion is a strictly stronger notion than language inclusion (see [8] for an example).

Downward Simulation on Tree Automata. Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. A *downward simulation* D is a binary relation on Q such that if $q\ D\ r$, then

1. $(q = \psi \implies r = \psi)$, and
2. for any $\langle q, a, q_1 \ldots q_n \rangle \in \delta$, there exists $\langle r, a, r_1 \ldots r_n \rangle \in \delta$ s.t. $q_i\, D\, r_i$ for $i : 1 \leq i \leq n$.

Since the set of all downward simulations on A is closed under union and under reflexive and transitive closure (cf. Lemma 4.1 in [19]), it follows that there is one unique maximal downward simulation on A, and that relation is a preorder. We call it *the downward simulation preorder on* A and write $\sqsubseteq^{\mathsf{dw}}$.

Downward Trace Inclusion on Tree Automata. Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. The *downward trace inclusion* preorder $\subseteq^{\mathsf{dw}}$ is a binary relation on Q s.t. $q \subseteq^{\mathsf{dw}} r$ iff for every tree $t \in \mathbb{C}(\Sigma)$ and for every t-run π_q with $\pi_q(\epsilon) = q$ there exists another t-run π_r s.t.

1. $\pi_r(\epsilon) = r$, and
2. $(\pi_q(v) = \psi \implies \pi_r(v) = \psi)$ for each leaf node $v \in dom(t)$.

Generally, one way of making downward language inclusion on the states of an automaton coincide with downward trace inclusion is by modifying the automaton to guarantee that **(1)** there is one unique final state which has no outgoing transitions, **(2)** from any other state, there is a path ending in that final state. Note that in a TDTA these two conditions are automatically satisfied: **(1)** since the final state is reached after reading a leaf of the tree, and **(2)** because only complete trees are in the language of the automaton. Thus, in a TDTA, downward language inclusion and downward trace inclusion coincide.

Backward Simulation on Word Automata. Let $A = (\Sigma, Q, \delta, I, F)$ be a NFA. A *backward simulation* B is a binary relation on Q s.t. if $q\ B\ r$, then

1. $(q \in F \implies r \in F)$ and $(q \in I \implies r \in I)$, and
2. for any $\langle q', a, q\rangle \in \delta$, there exists $\langle r', a, r\rangle \in \delta$ s.t. $q'\ B\ r'$.

Like for forward simulation, there is a unique maximal backward simulation on A, which is a preorder. We call it *the backward simulation preorder on* A and write $\sqsubseteq^{\mathsf{bw}}$.

Backward Trace Inclusion on Word Automata. Let $A = (\Sigma, Q, \delta, I, F)$ be a NFA and $w = \sigma_1\,\sigma_2 \ldots \sigma_n \in \Sigma^*$ a word of length n. A w-trace of A ending at q is a sequence of transitions $\pi = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q_n$ such that $q_n = q$. The *backward trace inclusion* preorder $\subseteq^{\mathsf{bw}}$ is a binary relation on Q such that $q \subseteq^{\mathsf{bw}} r$ iff

1. $(q \in F \implies r \in F)$ and $(q \in I \implies r \in I)$, and
2. for every word $w = \sigma_1\,\sigma_2 \ldots \sigma_n \in \Sigma^*$ and for every w-trace (ending at q) $\pi_q = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q$, there exists a w-trace (ending at r) $\pi_r = r_0 \xrightarrow{\sigma_1} r_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} r$ such that $(q_i \in F \implies r_i \in F \ \wedge\ q_i \in I \implies r_i \in I)$ for each $i : 1 \leq i \leq n$.

Upward Simulation on Tree Automata. Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. Given a binary relation R on Q, an *upward simulation* $U(R)$ induced by R is a binary relation on Q such that if $q\ U(R)\ r$, then

1. $(q = \psi \implies r = \psi)$ and $(q \in I \implies r \in I)$, and
2. for any $\langle q', a, q_1 \ldots q_n\rangle \in \delta$ with $q_i = q$ (for some $i : 1 \leq i \leq n$), there exists $\langle r', a, r_1 \ldots r_n\rangle \in \delta$ such that $r_i = r$, $q'\ U(R)\ r'$ and $q_j\ R\ r_j$ for each $j : 1 \leq j \neq i \leq n$.

Similarly to the case of downward simulation, for any given relation R, there is a unique maximal upward simulation induced by R which is a preorder (cf. Lemma 4.2 in [19]). We call it *the upward simulation preorder on A induced by R* and write $\sqsubseteq^{\mathsf{up}}(R)$.

Upward Trace Inclusion on Tree Automata. Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. Given a binary relation R on Q, the *upward trace inclusion preorder* $\subseteq^{\mathsf{up}}(R)$ *induced by* R is a binary relation on Q such that $q \subseteq^{\mathsf{up}}(R)\, r$ iff $(q = \psi \implies r = \psi)$ and the following holds: for every tree $t \in T(\Sigma)$ and for every t-run π_q with $\pi_q(v) = q$ for some leaf v of t, there exists a t-run π_r s.t.

1. $\pi_r(v) = r$,
2. for all prefixes v' of v, $(\pi_q(v') \in I \implies \pi_r(v') \in I)$, and
3. if $v'x \in dom(\pi_q)$, for some strict prefix v' of v and some $x \in \mathbb{N}$ s.t. $v'x$ is not a prefix of v, then $\pi_q(v'x)\ R\ \pi_r(v'x)$.

Downward trace inclusion is EXPTIME-complete for TDTA [14], while forward trace inclusion is PSPACE-complete for word automata. The complexity of upward trace inclusion depends on the relation R (e.g., it is PSPACE-complete for $R = id$). In contrast, downward/upward simulation preorder is computable in polynomial time [1], but typically yields only small under-approximations of the corresponding trace inclusions.

4 Transition Pruning Techniques

We define pruning relations on a TDTA $A = (\Sigma, Q, \delta, I)$. The intuition is that certain transitions may be deleted without changing the language, because 'better' transitions remain. We perform this pruning (i.e., deletion) of transitions by comparing their endpoints over the same symbol $\sigma \in \Sigma$. Given two binary relations R_{u} and R_{d} on Q, we define the following relation to compare transitions.

$$P(R_{\mathsf{u}}, R_{\mathsf{d}}) = \{(\langle p, \sigma, r_1 \cdots r_n\rangle, \langle p', \sigma, r'_1 \cdots r'_n\rangle) \mid p\ R_{\mathsf{u}}\ p' \text{ and } (r_1 \cdots r_n)\ \hat{R}_{\mathsf{d}}\ (r'_1 \cdots r'_n)\},$$

where $\hat{R}_{\mathsf{d}}$ results from lifting $R_{\mathsf{d}} \subseteq Q \times Q$ to $\hat{R}_{\mathsf{d}} \subseteq Q^n \times Q^n$, as defined below. The function P is monotone in the two arguments. If $t\, P\, t'$ then t may be pruned because t' is 'better' than t. We want $P(R_{\mathsf{u}}, R_{\mathsf{d}})$ to be a strict partial order (p.o.), i.e., irreflexive and transitive (and thus acyclic). There are two cases in which $P(R_{\mathsf{u}}, R_{\mathsf{d}})$ is guaranteed to be a strict p.o.: **(1)** R_{u} is some strict p.o. $<_{\mathsf{u}}$ and $\hat{R}_{\mathsf{d}}$ is the standard lifting $\hat{\leq}_{\mathsf{d}}$ of some p.o. $\leq_{\mathsf{d}}$ to tuples. I.e., $(r_1 \cdots r_n)\hat{\leq}_{\mathsf{d}}(r'_1 \cdots r'_n)$ iff $\forall_{1\leq i\leq n}.\, r_i \leq_{\mathsf{d}} r'_i$. The transitions in each pair of $P(<_{\mathsf{u}}, \leq_{\mathsf{d}})$ depart from different states and therefore the transitions are necessarily different. **(2)** R_{u} is some p.o. $\leq_{\mathsf{u}}$ and $\hat{R}_{\mathsf{d}}$ is the lifting $\hat{<}_{\mathsf{d}}$ of some strict p.o. $<_{\mathsf{d}}$ to tuples (defined below). In this case the transitions in each pair of $P(\leq_{\mathsf{u}}, <_{\mathsf{d}})$ may have the same origin but must go to different tuples of states. Since for two tuples $(r_1 \cdots r_n)$ and $(r'_1 \cdots r'_n)$ to be different it suffices that $r_i \neq r'_i$ for some $1 \leq i \leq n$, we define $\hat{<}_{\mathsf{d}}$ as a binary relation such that $(r_1 \cdots r_n)\hat{<}_{\mathsf{d}}(r'_1 \cdots r'_n)$ iff $\forall_{1\leq i\leq n}.\, r_i \leq_{\mathsf{d}} r'_i$, and $\exists_{1\leq i\leq n}.\, r_i <_{\mathsf{d}} r'_i$.

Let $A = (\Sigma, Q, \delta, I)$ be a TDTA and let $P \subseteq \delta \times \delta$ be a strict partial order. The pruned automaton is defined as $Prune(A, P) = (\Sigma, Q, \delta', I)$ where $\delta' = \{(p, \sigma, r) \in \delta \mid \nexists (p', \sigma, r') \in \delta.\, (p, \sigma, r)\, P\, (p', \sigma, r')\}$. Note that the pruned automaton $Prune(A, P)$ is unique. The transitions are removed without requiring the re-computation of the relation P, which could be expensive. Since removing transitions cannot introduce new trees in the language, $L(Prune(A, P)) \subseteq L(A)$. If the reverse inclusion holds too (so that the language is preserved), we say that P is *good for pruning* (GFP), i.e., P is GFP iff $L(Prune(A, P)) = L(A)$.

We now provide a complete picture of which combinations of simulation and trace inclusion relations are GFP. Recall that simulations are denoted by square symbols $\sqsubseteq$ while trace inclusions are denoted by round symbols $\subseteq$. For every partial order R, the corresponding strict p.o. is defined as $R \backslash R^{-1}$.

$P(\subset^{\mathsf{bw}}, \subset^{\mathsf{di}})$ is not GFP for word automata (see Fig. 2(a) in [13] for a counterexample). As mentioned before, words correspond to linear trees. Thus $P(\subset^{\mathsf{up}}(R), \subset^{\mathsf{dw}})$ is not GFP for tree automata (regardless of the relation R). Figure 1 presents several more counterexamples. For word automata, $P(\subset^{\mathsf{bw}}, \sqsubseteq^{\mathsf{di}})$ and $P(\sqsubseteq^{\mathsf{bw}}, \subset^{\mathsf{di}})$ are not GFP (Fig. 1b and c) even though $P(\subseteq^{\mathsf{bw}}, \sqsubset^{\mathsf{di}})$ and $P(\sqsubset^{\mathsf{bw}}, \subseteq^{\mathsf{di}})$ are (cf. [13]). Thus $P(\subset^{\mathsf{up}}(R), \sqsubseteq^{\mathsf{dw}})$ and $P(\sqsubseteq^{\mathsf{up}}(R), \subset^{\mathsf{dw}})$ are not GFP for tree automata (regardless of the relation R). For tree automata, $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{dw}}), id)$ and $P(\sqsubset^{\mathsf{up}}(\subset^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$ are not GFP (Fig. 1a and d). Moreover, a complex counterexample (see [8]) is needed to show that $P(\sqsubset^{\mathsf{up}}(\sqsubset^{\mathsf{dw}}), \subset^{\mathsf{dw}})$ is not GFP.

The following theorems and corollaries provide several relations which are GFP.

Theorem 1. *For every strict partial order $R \subset \subseteq^{\mathsf{dw}}$, it holds that $P(id, R)$ is GFP.*

Corollary 1. *By Theorem 1, $P(id, \subset^{\mathsf{dw}})$ and $P(id, \sqsubset^{\mathsf{dw}})$ are GFP.*

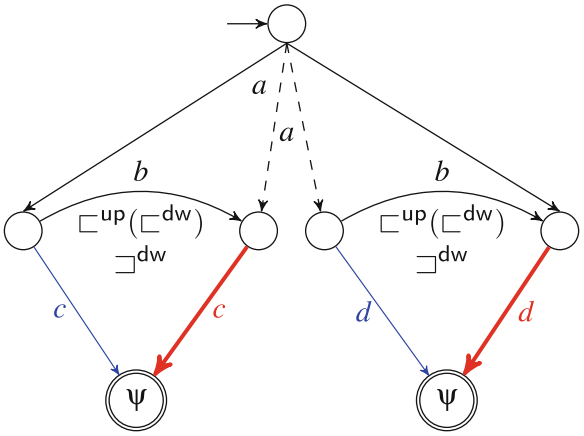
Theorem 2. *For every strict partial order $R \subset \subseteq^{\mathsf{up}}(id)$, it holds that $P(R, id)$ is GFP.*

Corollary 2. *By Theorem 2, $P(\subset^{\mathsf{up}}(id), id)$ and $P(\sqsubset^{\mathsf{up}}(id), id)$ are GFP.*

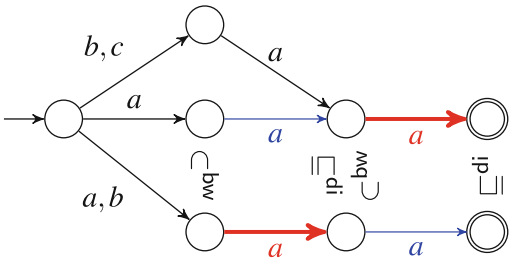
Definition 1. *Given a tree automaton A, a binary relation W on its states is called a* downup-relation *iff the following condition holds: If $p\ W\ q$ then for every tree $t \in \mathbb{T}(\Sigma)$ and accepting t-run π from p there exists an accepting t-run π' from q such that $\forall_{v \in \mathbb{N}^*}\ \pi(v) \sqsubseteq^{\mathsf{up}}(W)\ \pi'(v)$.*

Lemma 1. *Any relation V satisfying **(1)** V is a downward simulation, and **(2)** $id \subseteq V \subseteq \sqsubseteq^{\mathsf{up}}(V)$ is a downup-relation. In particular, id is a downup-relation, but $\sqsubseteq^{\mathsf{dw}}$ and $\sqsubseteq^{\mathsf{up}}(id)$ are not.*

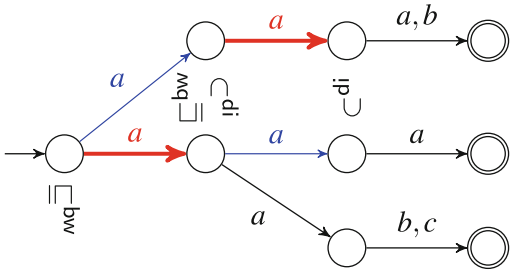
Theorem 3. *For every downup-relation W, it holds that $P(\sqsubset^{\mathsf{up}}(W), \subseteq^{\mathsf{dw}})$ is GFP.*



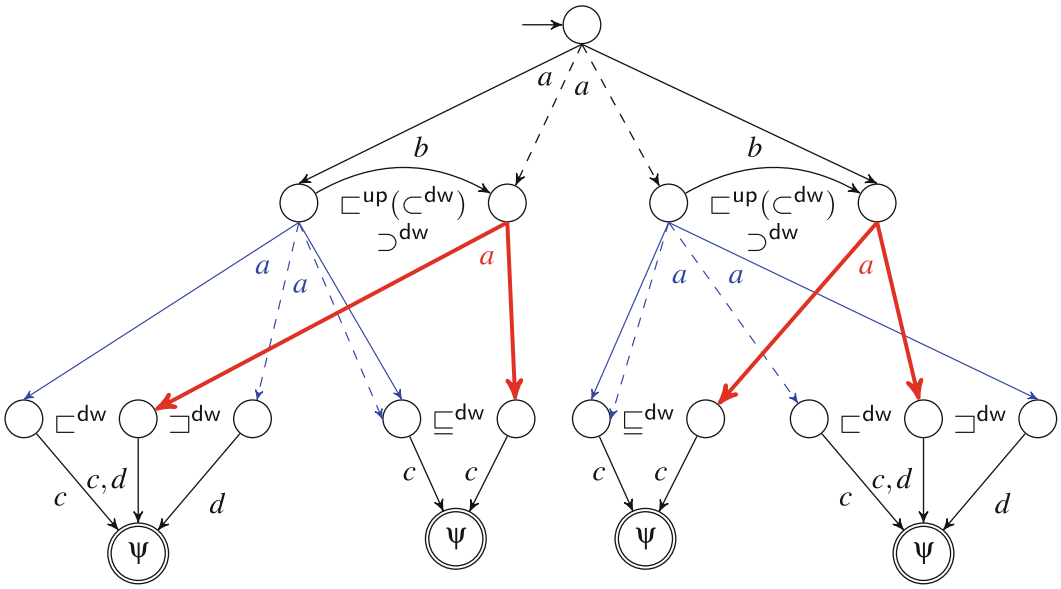
(a) $P(\sqsubset^{up}(\sqsubset^{dw}), id)$ is not GFP: if we remove the blue transitions, the automaton no longer accepts the tree $a(c,d)$. We are considering $\Sigma_0 = \{c,d\}$, $\Sigma_1 = \{b\}$ and $\Sigma_2 = \{a\}$.



(b) $P(\subset^{bw}, \sqsubseteq^{di})$ is not GFP for words: if we remove the blue transitions, the automaton no longer accepts the word *aaa*.



(c) $P(\sqsubseteq^{bw}, \subset^{di})$ is not GFP for words: if we remove the blue transitions, the automaton no longer accepts the word *aaa*.



(d) $P(\sqsubset^{up}(\subset^{dw}), \sqsubset^{dw})$ is not GFP: if we remove the blue transitions, the tree $a(a(c,c),a(c,c))$ is no longer accepted. We are considering $\Sigma_0 = \{c,d\}$, $\Sigma_1 = \{b\}$ and $\Sigma_2 = \{a\}$.

Fig. 1. GFP counterexamples. A transition is drawn in dashed when a different transition by the same symbol departing from the same state already exists. We draw a transition in thick red when it is better than another transition (drawn in thin blue).

Proof. Let $A' = Prune(A, P(\sqsubset^{\mathsf{up}}(W), \subseteq^{\mathsf{dw}}))$. We show $L(A) \subseteq L(A')$. If $t \in L(A)$ then there exists an accepting t-run $\hat{\pi}$ in A. We show that there is an accepting t-run $\hat{\pi}'$ in A'.

For each accepting t-run π in A, let $level_i(\pi)$ be the tuple of states that π visits at depth i in the tree, read from left to right. Formally, let $(x_1, \ldots, x_k)$ with $x_j \in \mathbb{N}^i$ be the set of all tree positions of depth i s.t. $x_j \in dom(\pi)$, in lexicographically increasing order. Then $level_i(\pi) = (\pi(x_1), \ldots, \pi(x_k)) \in Q^k$. By lifting partial orders on Q to partial orders on tuples, we can compare such tuples w.r.t. $\sqsubseteq^{\mathsf{up}}(W)$. We say that an accepting t-run π is i-good iff it does not contain any transition from $A - A'$ from any position $v \in \mathbb{N}^*$ with $|v| < i$. I.e., no pruned transition is used in the first i levels of the tree.

We now define a strict partial order $<_i$ on the set of accepting t-runs in A. Let $\pi <_i \pi'$ iff $\exists k \leq i.\ level_k(\pi) \sqsubset^{\mathsf{up}}(W)\ level_k(\pi')$ and $\forall l < k.\ level_l(\pi) \sqsubseteq^{\mathsf{up}}(W)\ level_l(\pi')$. Note that $<_i$ only depends on the first i levels of the run. Given A, t and i, there are only finitely many different such i-prefixes of accepting t-runs. By our assumption that $\hat{\pi}$ is an accepting t-run in A, the set of accepting t-runs in A is non-empty. Thus, for any i, there must exist some accepting t-run π in A that is maximal w.r.t. $<_i$.

We now show that this π is also i-good, by assuming the contrary and deriving a contradiction. Suppose that π is not i-good. Then it must contain a transition $\langle p, \sigma, r_1 \cdots r_n \rangle$ from $A - A'$ used at the root of some subtree t' of t at some level $j < i$. Since $A' = Prune(A, P(\sqsubset^{\mathsf{up}}(W), \subseteq^{\mathsf{dw}}))$, there must exist another transition $\langle p', \sigma, r'_1 \cdots r'_n \rangle$ in A' s.t. (1) $(r_1, \ldots, r_n) \subseteq^{\mathsf{dw}} (r'_1, \ldots, r'_n)$ and (2) $p \sqsubset^{\mathsf{up}}(W) p'$.

First consider the implications of (2). Upward simulation propagates upward stepwise (though only in non-strict form after the first step). So p' can imitate the upward path of p to the root of t, maintaining $\sqsubseteq^{\mathsf{up}}(W)$ between the corresponding states. The states on side branches joining in along the upward path from p can be matched by W-larger states in joining side branches along the upward path from p'. From Definition 1 we obtain that these W-larger states in p's joining side branches can accept their subtrees of t via computations that are everywhere $\sqsubseteq^{\mathsf{up}}(W)$ larger than corresponding states in computations from ps joining side branches. So there must be an accepting run π' on t s.t. (3) π' is at state p' at the root of t' and uses transition $\langle p', \sigma, r'_1 \cdots r'_n \rangle$ from p', and (4) for all $v \in \mathbb{N}^*$ where $t(v) \notin t'$ we have $\pi(v) \sqsubseteq^{\mathsf{up}}(W) \pi'(v)$. Moreover, by conditions (1) and (3), π' can be extended from $r'_1, \ldots, r'_n$ to accept also the subtree t'. Thus π' is an accepting t-run in A. By conditions (2) and (4) we obtain that $\forall l \leq j.\ level_l(\pi) \sqsubseteq^{\mathsf{up}}(W)\ level_l(\pi')$. By (2) we get even $level_j(\pi) \sqsubset^{\mathsf{up}}(W)\ level_j(\pi')$ and thus $\pi <_j \pi'$. Since $j < i$ we also have $\pi <_i \pi'$ and thus π was not maximal w.r.t. $<_i$. Contradiction. So we have shown that for every $t \in L(A)$ there exists an i-good accepting run for every finite i.

If $t \in L(A)$ then there exists an accepting t-run $\hat{\pi}$ in A. Then there exists an accepting t-run $\hat{\pi}'$ that is i-good, where i is the height of t. Thus $\hat{\pi}'$ is a run in A' and $t \in L(A')$. □

Corollary 3. *It follows from Lemma 1 and from the fact that GFP is downward closed that* $P(\sqsubset^{\mathsf{up}}(V), \subseteq^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(V), \subset^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(V), \sqsubseteq^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(V), \sqsubset^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(V), id)$, $P(\sqsubset^{\mathsf{up}}(id), \subseteq^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(id), \subset^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(id), \sqsubseteq^{\mathsf{dw}})$ *and* $P(\sqsubset^{\mathsf{up}}(id), \sqsubset^{\mathsf{dw}})$ *are GFP.*

Theorem 4. $P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$ *is GFP.*

Proof. Let $A' = Prune(A, P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}}))$. We show $L(A) \subseteq L(A')$. If $t \in L(A)$ then there exists an accepting t-run $\hat{\pi}$ in A. We show that there is an accepting t-run $\hat{\pi}'$ in A'.

For each accepting t-run π in A, let $level_i(\pi)$ be the tuple of states that π visits at depth i in the tree, read from left to right. Formally, let $(x_1, \ldots, x_k)$ with $x_j \in \mathbb{N}^i$ be the set of all tree positions of depth i s.t. $x_j \in dom(\pi)$, in lexicographically increasing order. Then $level_i(\pi) = (\pi(x_1), \ldots, \pi(x_k)) \in Q^k$. By lifting partial orders on Q to partial orders on tuples we can compare such tuples w.r.t. $\sqsubseteq^{\mathsf{dw}}$. We say that an accepting t-run π is i-good if it does not contain any transition from $A - A'$ from any position $v \in \mathbb{N}^*$ with $|v| < i$. I.e., no pruned transitions are used in the first i levels of the tree.

We now show, by induction on i, the following property (C): For every i and every accepting t-run π in A there exists an i-good accepting t-run π' in A s.t. $level_i(\pi) \sqsubseteq^{\mathsf{dw}} level_i(\pi')$.

The base case is $i = 0$. Every accepting t-run π in A is trivially 0-good itself and thus satisfies (C).

For the induction step, let S be the set of all $(i-1)$-good accepting t-runs π' in A s.t. $level_{i-1}(\pi) \sqsubseteq^{\mathsf{dw}} level_{i-1}(\pi')$. Since π is an accepting t-run, by induction hypothesis, S is non-empty. Let $S' \subseteq S$ be the subset of S containing exactly those runs $\pi' \in S$ that additionally satisfy $level_i(\pi) \sqsubseteq^{\mathsf{dw}} level_i(\pi')$. From $level_{i-1}(\pi) \sqsubseteq^{\mathsf{dw}} level_{i-1}(\pi')$ and the fact that $\sqsubseteq^{\mathsf{dw}}$ is preserved downward-stepwise, we obtain that S' is non-empty. Now we can select some $\pi' \in S'$ s.t. $level_i(\pi')$ is maximal, w.r.t. $\sqsubseteq^{\mathsf{dw}}$, relative to the other runs in S'. We claim that π' is i-good and $level_i(\pi) \sqsubseteq^{\mathsf{dw}} level_i(\pi')$. The second part of this claim holds because $\pi' \in S'$.

We show that π' is i-good by contraposition. Suppose that π' is not i-good. Then it must contain a transition $\langle p, \sigma, r_1 \cdots r_n \rangle$ from $A - A'$. Since π' is $(i-1)$-good, this transition must start at depth $(i-1)$ in the tree. Since $A' = Prune(A, P(\subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}}))$, there must exist another transition $\langle p', \sigma, r'_1 \cdots r'_n \rangle$ in A' s.t. $p \subseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}) p'$ and $(r_1, \ldots, r_n) \sqsubset^{\mathsf{dw}} (r'_1, \ldots, r'_n)$. From the definition of $\subseteq^{\mathsf{up}}$ $(\sqsubseteq^{\mathsf{dw}})$ we obtain that there exists another accepting t-run π_1 in A (that uses the transition $\langle p', \sigma, r'_1 \cdots r'_n \rangle$) s.t. $level_i(\pi') \sqsubset^{\mathsf{dw}} level_i(\pi_1)$. The run π_1 is not necessarily i-good or $(i-1)$-good. However, by induction hypothesis, there exists some accepting t-run π_2 in A that is $(i-1)$-good and satisfies $level_{i-1}(\pi_1) \sqsubseteq^{\mathsf{dw}} level_{i-1}(\pi_2)$. Since $\sqsubseteq^{\mathsf{dw}}$ is preserved stepwise, there also exists an accepting t-run π_3 in A (that coincides with π_2 up-to depth $(i-1)$), which is $(i-1)$-good and satisfies $level_i(\pi_1) \sqsubseteq^{\mathsf{dw}} level_i(\pi_3)$. In particular, $\pi_3 \in S'$.

From $level_i(\pi') \sqsubset^{\mathsf{dw}} level_i(\pi_1)$ and $level_i(\pi_1) \sqsubseteq^{\mathsf{dw}} level_i(\pi_3)$ we obtain $level_i(\pi') \sqsubset^{\mathsf{dw}} level_i(\pi_3)$. This contradicts our condition above that π' must

be $level_i$ maximal w.r.t. $\sqsubseteq^{\mathsf{dw}}$ in S'. This concludes the induction step and the proof of property (C).

If $t \in L(A)$ then there exists an accepting t-run $\hat{\pi}$ in A. By property (C), there exists an accepting t-run $\hat{\pi}'$ that is i-good, where i is the height of t. Therefore $\hat{\pi}'$ does not use any transition from $A - A'$ and is thus also a run in A'. So we obtain $t \in L(A')$. □

Corollary 4. *It follows from Theorem 4 and the fact that GFP is downward closed that* $P(\subset^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, $P(\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, $P(\sqsubset^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, $P(\subseteq^{\mathsf{up}}(id), \sqsubset^{\mathsf{dw}})$, $P(\subset^{\mathsf{up}}(id), \sqsubset^{\mathsf{dw}})$, $P(\sqsubseteq^{\mathsf{up}}(id), \sqsubset^{\mathsf{dw}})$ *and* $P(id, \sqsubset^{\mathsf{dw}})$ *are GFP.*

5 State Quotienting Techniques

A classic method for reducing the size of automata is state quotienting. Given a suitable equivalence relation on the set of states, each equivalence class is collapsed into just one state. From a preorder $\sqsubseteq$ one obtains an equivalence relation $\equiv := \sqsubseteq \cap \sqsupseteq$. We now define quotienting w.r.t. $\equiv$. Let $A = (\Sigma, Q, \delta, I)$ be a TDTA and let $\sqsubseteq$ be a preorder on Q. Given $q \in Q$, we denote by $[q]$ its equivalence class w.r.t $\equiv$. For $P \subseteq Q$, $[P]$ denotes the set of equivalence classes $[P] = \{[p] \,|\, p \in P\}$. We define the quotient automaton w.r.t. $\equiv$ as $A/\equiv := (\Sigma, [Q], \delta_{A/\equiv}, [I])$, where $\delta_{A/\equiv} = \{\langle [q], \sigma, [q_1] \dots [q_n] \rangle \mid \langle q, \sigma, q_1 \dots q_n \rangle \in \delta_A\}$. It is trivial that $L(A) \subseteq L(A/\equiv)$ for any $\equiv$. If the reverse inclusion also holds, i.e., if $L(A) = L(A/\equiv)$, we say that $\equiv$ is *good for quotienting* (GFQ).

It was shown in [19] that $\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}$ and $\sqsubseteq^{\mathsf{up}}(id) \cap \sqsupseteq^{\mathsf{up}}(id)$ are GFQ. Here we generalize this result from simulation to trace equivalence. Let $\equiv^{\mathsf{dw}} := \subseteq^{\mathsf{dw}} \cap \supseteq^{\mathsf{dw}}$ and $\equiv^{\mathsf{up}}(R) := \subseteq^{\mathsf{up}}(R) \cap \supseteq^{\mathsf{up}}(R)$.

Theorem 5. $\equiv^{\mathsf{dw}}$ *is GFQ.*

Theorem 6. $\equiv^{\mathsf{up}}(id)$ *is GFQ.*

In [8] we present a counterexample showing that $\equiv := \sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}}) \cap \sqsupseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}} \cap \sqsupseteq^{\mathsf{dw}})$ is not GFQ. This is an adaptation from the Example 5 in [19], where the inducing relation is referred to as the *downward bisimulation equivalence* and the automata are seen bottom-up.

One of the best methods previously known for reducing TA performs state quotienting based on a combination of downward and upward simulation [4]. However, this method cannot achieve any further reduction on an automaton which has been previously reduced with the techniques we described above [8].

6 Lookahead Simulations

Simulation preorders are generally not very good under-approximations of trace inclusion, since they are much smaller on many automata. Thus we consider better approximations that are still efficiently computable.

For word automata, more general *lookahead simulations* were introduced in [13]. These provide a practically useful tradeoff between the computational effort and the size of the obtained relations. Lookahead simulations can also be seen as a particular restriction of the more general (but less practically useful) *multipebble simulations* [17]. We generalize lookahead simulations to tree automata in order to compute good under-approximations of trace inclusions.

Intuition by Simulation Games. Normal simulation preorder on labeled transition graphs can be characterized by a game between two players, Spoiler and Duplicator. Given a pair of states (q_0, r_0), Spoiler wants to show that (q_0, r_0) is not contained in the simulation preorder relation, while Duplicator has the opposite goal. Starting in the initial configuration (q_0, r_0), Spoiler chooses a transition $q_0 \xrightarrow{\sigma} q_1$ and Duplicator must imitate it *stepwise* by choosing a transition with the same symbol $r_0 \xrightarrow{\sigma} r_1$. This yields a new configuration (q_1, r_1) from which the game continues. If a player cannot move the other wins. Duplicator wins every infinite game. Simulation holds iff Duplicator wins.

In normal simulation, Duplicator only knows Spoiler's very next step (see above), while in *k-lookahead simulation* Duplicator knows Spoiler's k next steps in advance (unless Spoiler's move ends in a deadlocked state - i.e., a state with no transitions). As the parameter k increases, the k-lookahead simulation relation becomes larger and thus approximates the trace inclusion relation better and better. Trace inclusion can also be characterized by a game. In the trace inclusion game, Duplicator knows *all* steps of Spoiler in the entire game in advance.

For every fixed k, k-lookahead simulation is computable in polynomial time, though the complexity rises quickly in k: it is doubly exponential for downward- and single exponential for upward lookahead simulation (due to the downward branching of trees). A crucial trick makes it possible to practically compute it for nontrivial k: Spoiler's moves are built incrementally, and Duplicator need not respond to all of Spoiler's announced k next steps, but only to a prefix of them, after which he may request fresh information [13]. Thus Duplicator just uses the minimal lookahead necessary to win the current step.

Lookahead Downward Simulation. We say that a tree t is k-bounded iff for all leaves v of t, either **(a)** $|v| = k$, or **(b)** $|v| < k$ and v is closed. Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. A *k-lookahead downward simulation* $L^{k-\mathsf{dw}}$ is a binary relation on Q such that if $q\ L^{k-\mathsf{dw}}\ r$, then $(q = \psi \implies r = \psi)$ and the following holds: Let π_k be a run on a k-bounded tree t_k with $\pi(\epsilon) = q$ s.t. every leaf node of π_k is either at depth k or downward-deadlocked (i.e., no more downward transitions exist). Then there must exist a run π'_k over a nonempty prefix t'_k of t_k s.t. (1) $\pi'_k(\epsilon) = r$, and (2) for every leaf v of π'_k, $\pi_k(v)\ L^{k-\mathsf{dw}}\ \pi'_k(v)$. Since, for given A and $k \geq 1$, lookahead downward simulations are closed under union, there exists a unique maximal one that we call *the k-lookahead downward simulation on A*, denoted by $\sqsubseteq^{k\text{-}\mathsf{dw}}$. While $\sqsubseteq^{k\text{-}\mathsf{dw}}$ is trivially reflexive, it is not transitive in general (cf. [13], Appendix B). Since we only use it as a means to under-approximate the transitive trace inclusion relation $\subseteq^{\mathsf{dw}}$

(and require a preorder to induce an equivalence), we work with its transitive closure $\preceq^{k\text{-dw}} := (\sqsubseteq^{k\text{-dw}})^+$. In particular, $\preceq^{k\text{-dw}} \subseteq \subseteq^{\mathsf{dw}}$.

Lookahead Upward Simulation. Let $A = (\Sigma, Q, \delta, I)$ be a TDTA. A *k-lookahead upward simulation* on A induced by a relation R is a binary relation $L^{k-\mathsf{up}}(R)$ on Q s.t. if $q\ L^{k-\mathsf{up}}(R)\ r$, then $(q = \psi \implies r = \psi)$ and the following holds: Let π be a run over a tree $t \in \mathbb{T}(\Sigma)$ with $\pi(v) = q$ for some bottom leaf v s.t. either $|v| = k$ or $0 < |v| < k$ and $\pi(\epsilon)$ is upward-deadlocked (i.e., no more upward transitions exist).

Then there must exist v', v'' such that $v = v'v''$ and $|v''| \geq 1$ and a run π' over $t_{v'}$ s.t. the following holds. (1) $\pi'(v'') = r$, (2) $\pi(v')\ L^{k-\mathsf{up}}(R)\ \pi'(\epsilon)$, (3) $\pi(v'x) \in I \implies \pi'(x) \in I$ for all prefixes x of v'', (4) If $v'xy \in dom(\pi)$ for some strict prefix x of v'' and some $y \in \mathbb{N}$ where xy is not a prefix of v'' then $\pi(v'xy)\ R\ \pi'(xy)$.

Since, for given A, $k \geq 1$ and R, lookahead upward simulations are closed under union, there exists a unique maximal one that we call *the k-lookahead upward simulation induced by R on A*, denoted by $\sqsubseteq^{k\text{-up}}(R)$. Since both R and $\sqsubseteq^{k\text{-up}}(R)$ are not necessarily transitive, we first compute its transitive closure, R^+, and we then compute $\preceq^{k\text{-up}}(R) := (\sqsubseteq^{k\text{-up}}(R^+))^+$, which under-approximates the upward trace inclusion $\subseteq^{\mathsf{up}}(R^+)$.

7 Experiments

Our tree automata reduction algorithm (tool available [7]) combines transition pruning techniques (Sect. 4) with quotienting techniques (Sect. 5). Trace inclusions are under-approximated by lookahead simulations (Sect. 6) where higher lookaheads are harder to compute but yield better approximations. The parameters $x, y \geq 1$ describe the lookahead for downward/upward lookahead simulations, respectively. Downward lookahead simulation is harder to compute than upward lookahead simulation, since the number of possible moves is doubly exponential in x (due to the downward branching of the tree) while for upward-simulation it is only single exponential in y. We use (x, y) as $(1, 1)$, $(2, 4)$ and $(3, 7)$.

Besides pruning and quotienting, we also use the operation RU that removes useless states, i.e., states that either cannot be reached from any initial state or from which no tree can be accepted. Let $Op(x, y)$ be the following sequence of operations on tree automata: RU, quotienting with $\preceq^{x\text{-dw}}$, pruning with $P(id, \prec^{x\text{-dw}})$, RU, quotienting with $\preceq^{y\text{-up}}(id)$, pruning with $P(\prec^{y\text{-up}}(id), id)$, pruning with $P(\sqsubset^{\mathsf{up}}(id), \preceq^{x\text{-dw}})$, RU, quotienting with $\preceq^{y\text{-up}}(id)$, pruning with $P(\preceq^{y\text{-up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, RU. It is language preserving by the Theorems of Sects. 4 and 5. The order of the operations is chosen according to some considerations of efficiency. (No order is ideal for all instances.)

Our algorithm $Heavy(1, 1)$ just iterates $Op(1, 1)$ until a fixpoint is reached. For efficiency reasons, the general algorithm $Heavy(x, y)$ does not iterate $Op(x, y)$, but uses a double loop: it iterates the sequence $Heavy(1, 1)\,Op(x, y)$ until a fixpoint is reached.

We compare the reduction performance of several algorithms.

RU: RU. (Previously present in `libvata`.)
RUQ: RU and quotienting with $\sqsubseteq^{\mathsf{dw}}$. (Previously present in `libvata`.)
RUQP: RUQ, plus pruning with $P(id, \sqsubset^{\mathsf{dw}})$. (Not in `libvata`, but simple.)
Heavy: $Heavy(1,1)$, $Heavy(2,4)$ and $Heavy(3,7)$. (New.)

We tested these algorithms on three sets of automata from the `libvata` distribution. The first set are 27 moderate-sized automata (87 states and 816 transitions on avg.) derived from regular model checking applications. Heavy(1,1), on avg., reduced the number of states and transitions *to* 27 % and 14 % of the original sizes, resp. (Note the difference between 'to' and 'by'.) In contrast, RU did not perform any reduction in any case, RUQ, on avg., reduced the number of states and transitions only to 81 % and 80 % of the original sizes and RUQP reduced the number of states and transitions to 81 % and 32 % of the original sizes; cf. Fig. 2. The average computation times of Heavy(1,1), RUQP, RUQ and RU were, respectively, 0.05 s, 0.03 s, 0.006 s and 0.001 s.

The second set are 62 larger automata (586 states and 8865 transitions, on avg.) derived from regular model checking applications. Heavy(1,1), on avg., reduced the number of states and transitions *to* 4.2 % and 0.7 % of the original sizes. In contrast, RU did not perform any reduction in any case, RUQ, on avg., reduced the number of states and transitions to 75.2 % and 74.8 % of the original sizes and RUQP reduced the number of states and transitions to 75.2 % and 15.8 % of the original sizes [8]. The average computation times of Heavy(1,1), RUQP, RUQ and RU were, respectively, 2.7 s, 2.1 s, 0.2 s and 0.02 s.

The third set are 14,498 automata (57 states and 266 transitions on avg.) from the shape analysis tool Forester [23]. Heavy(1,1), on avg., reduced the number of states/transitions *to* 76.4 % and 67.9 % of the original, resp. RUQ and RUQP reduced the states and transitions only to 94 % and 88 %, resp. The average computation times of Heavy(1,1), RUQP, RUQ and RU were, respectively, 0.21 s, 0.014 s, 0.004 s, and 0.0006 s.

Due to the particular structure of the automata in these 3 sample sets, $Heavy(2,4)$ and $Heavy(3,7)$ had hardly any advantage over $Heavy(1,1)$. However, in general they can perform significantly better.

We also tested the algorithms on randomly generated tree automata, according to a generalization of the Tabakov-Vardi model of random word automata [24]. Given parameters n, s, td (transition density) and ad (acceptance density), it generates tree automata with n states, s symbols (each of rank 2), $n * td$ randomly assigned transitions for each symbol, and $n * ad$ randomly assigned leaf rules. Figure 3 shows the results of reducing automata of varying td with different methods.

8 Summary and Conclusion

The tables in Figs. 4 and 5 summarize all our results on pruning and quotienting, respectively. Note that negative results propagate to larger relations and positive results propagate to smaller relations (i.e., GFP/GFQ is downward closed).

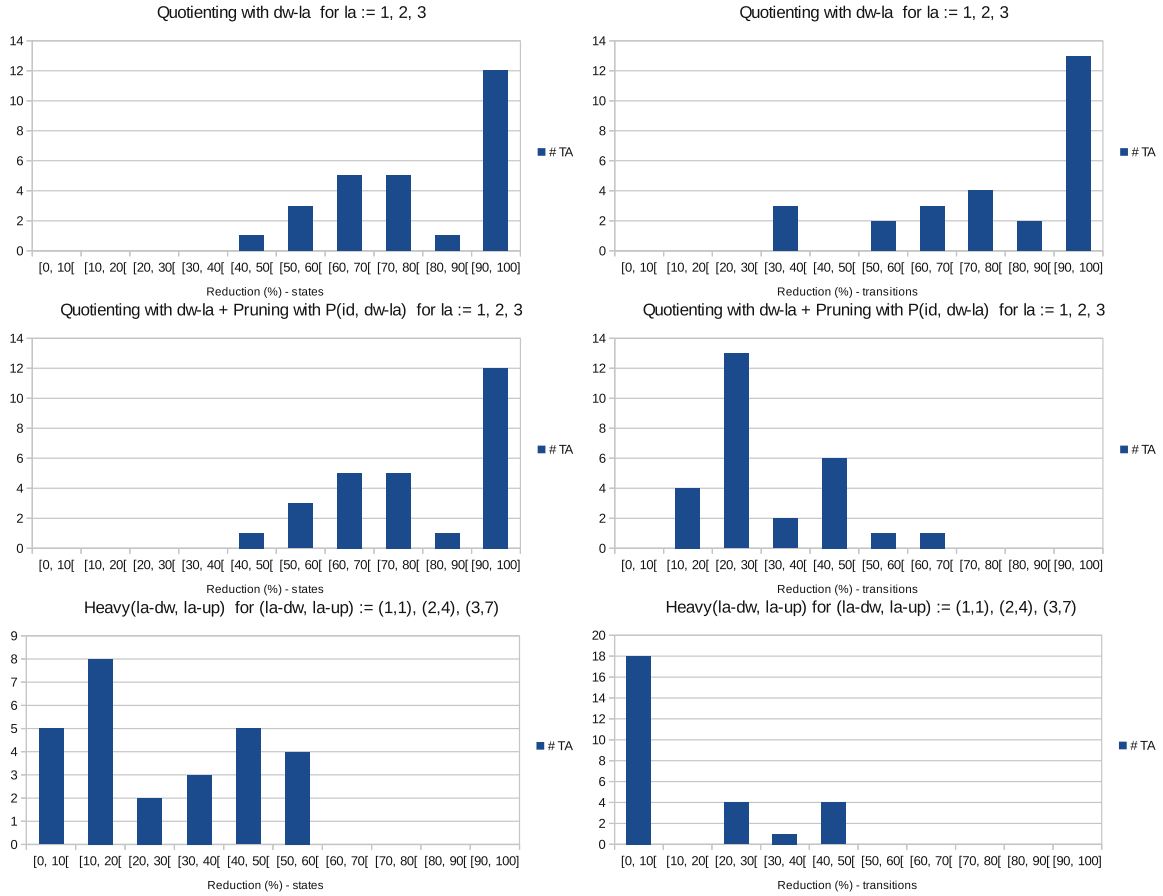


Fig. 2. Reduction of 27 moderate-sized tree automata by methods RUQ (top row), RUQP (middle row), and Heavy (bottom row). A bar of height h at an interval $[x, x{+}10[$ means that h of the 27 automata were reduced to a size between $x\,\%$ and $(x + 10)\,\%$ of their original size. The reductions in the numbers of states/transitions are shown on the left/right, respectively. On this set of automata, the methods Heavy(2,4) and Heavy(3,7) gave exactly the same results as Heavy(1,1).

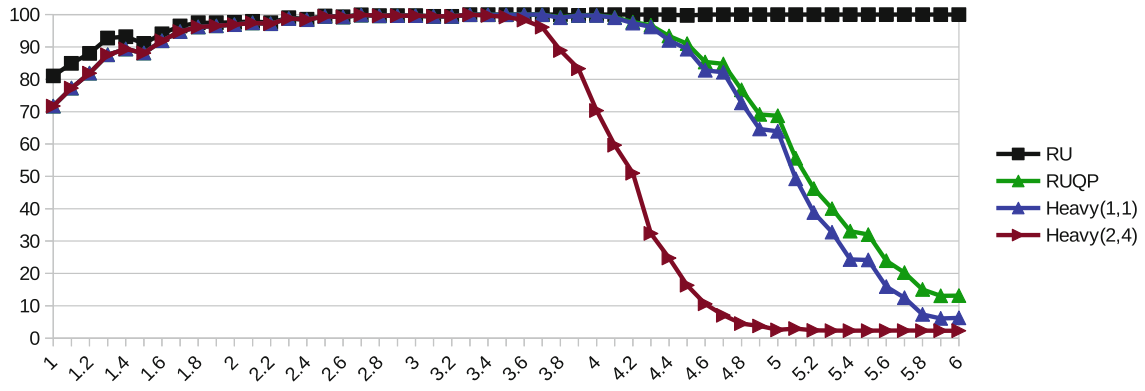


Fig. 3. Reduction of Tabakov-Vardi random tree automata with $n = 100, s = 2$ and $ad = 0.8$. The x-axis gives the transition density td, and the y-axis gives the average number of states after reduction with the various methods (smaller is better). Each data point is the average of 400 random automata. Note that Heavy(2,4) reduces much better than Heavy(1,1) for $td \geq 3.5$. Computing Heavy(x,y) for even higher x, y is very slow on (some instances of) random automata.

$R_u \backslash R_i$		R_d: id	$\sqsubset^{dw}$	$\sqsubseteq^{dw}$	$\subset^{dw}$	$\subseteq^{dw}$
id	id	–	✓	–	✓	–
$\sqsubset^{up}$	id	✓	✓	✓	✓	✓
	$\sqsubset^{dw}$	×	✓	×	×	×
	$\sqsubseteq^{dw}$	×	✓	×	×	×
	downup-rel.	✓	✓	✓	✓	✓
	$\subset^{dw}$	×	×	×	×	×
	$\subseteq^{dw}$	×	×	×	×	×
$\sqsubseteq^{up}$	id	–	✓	–	×	–
	$\sqsubset^{dw}$	–	✓	–	×	–
	$\sqsubseteq^{dw}$	–	✓	–	×	–
	$\subset^{dw}$	–	×	–	×	–
	$\subseteq^{dw}$	–	×	–	×	–
$\subset^{up}$	id	✓	✓	×	×	×
	$\sqsubset^{dw}$	×	✓	×	×	×
	$\sqsubseteq^{dw}$	×	✓	×	×	×
	$\subset^{dw}$	×	×	×	×	×
	$\subseteq^{dw}$	×	×	×	×	×
$\subseteq^{up}$	id	–	✓	–	×	–
	$\sqsubset^{dw}$	–	✓	–	×	–
	$\sqsubseteq^{dw}$	–	✓	–	×	–
	$\subset^{dw}$	–	×	–	×	–
	$\subseteq^{dw}$	–	×	–	×	–

Fig. 4. GFP relations $P(R_u(R_i), R_d)$ for tree automata. Relations which are GFP are marked with ✓, those which are not are marked with × and – is used to mark relations where the test does not apply due to them being reflexive (and therefore not asymmetric).

R	
$\subseteq^{dw}$	✓
$\sqsubseteq^{dw}$	✓
$\sqsubseteq^{up}$: id	✓
$\sqsubset^{dw}$	–
$\sqsubseteq^{dw}$	×
$\subset^{dw}$	–
$\subseteq^{dw}$	×
$\subseteq^{up}$: id	✓
$\sqsubset^{dw}$	–
$\sqsubseteq^{dw}$	×
$\subset^{dw}$	–
$\subseteq^{dw}$	×

Fig. 5. GFQ relations R for tree automata. Relations which are GFQ are marked with ✓ and those which are not are marked with ×. The relations marked with – are not even reflexive in general (unless all transitions are linear; in this case we have a word automaton and these relations are the same as $\sqsubseteq^{up}(id)$ and $\subseteq^{up}(id)$, respectively).

The experiments show that our Heavy(x,y) algorithm can significantly reduce the size of many classes of nondeterministic tree automata, and that it is sufficiently fast to handle instances with hundreds of states and thousands of transitions.

References

1. Abdulla, P.A., Bouajjani, A., Holík, L., Kaati, L., Vojnar, T.: Computing simulations over tree automata. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 93–108. Springer, Heidelberg (2008)
2. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)

3. Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 224–239. Springer, Heidelberg (2013)
4. Abdulla, P.A., Holík, L., Kaati, L., Vojnar, T.: A uniform (bi-)simulation-based framework for reducing tree automata. Electr. Notes Theor. Comput. Sci. **251**, 27–48 (2009)
5. Abdulla, P.A., Legay, A., d'Orso, J., Rezine, A.: Simulation-based iteration of tree transducers. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 30–44. Springer, Heidelberg (2005)
6. Abdulla, P.A., Legay, A., d'Orso, J., Rezine, A.: Tree regular model checking: a simulation-based approach. J. Log. Algebr. Program. **69**(1–2), 93–121 (2006)
7. Almeida, R., Holík, L., Mayr, R.: HeavyMinOTAut (2015). http://tinyurl.com/pm2b4qk
8. Almeida, R., Holík, L., Mayr, R.: Reduction of nondeterministic tree automata. Technical report EDI-INF-RR-1421, University of Edinburgh (2016). arXiv 1512.08823
9. Basin, D., Karlund, N., Møller, A.: Mona (2015). http://www.brics.dk/mona
10. Bonchi, F., Pous, D.: Checking NFA equivalence with bisimulations up to congruence. In: Principles of Programming Languages (POPL), Rome, Italy. ACM (2013)
11. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008)
12. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
13. Clemente, L., Mayr, R.: Advanced automata minimization. In: 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, pp. 63–74. ACM (2013)
14. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2008). http://www.grappa.univ-lille3.fr/tata. Release 18 November 2008
15. Durand, I.: Autowrite (2015). http://dept-info.labri.fr/~idurand/autowrite
16. Genet, T., et al.: Timbuk (2015). http://www.irisa.fr/celtique/genet/timbuk/
17. Etessami, K.: A hierarchy of polynomial-time computable simulations for automata. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 131–144. Springer, Heidelberg (2002)
18. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 424–440. Springer, Heidelberg (2011)
19. Holík, L.: Simulations and Antichains for Efficient Handling of Finite Automata. Ph.D. thesis, Faculty of Information Technology of Brno University of Technology (2011)
20. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 740–755. Springer, Heidelberg (2013)
21. Holík, L., Lengál, O., Šimáček, J., Vojnar, T.: Efficient inclusion checking on explicit and semi-symbolic tree automata. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 243–258. Springer, Heidelberg (2011)

22. Lengál, O., Simácek, J., Vojnar, T.: Libvata: highly optimised non-deterministic finite tree automata library (2015). http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/
23. Lengál, O., Simácek, J., Vojnar, T., Habermehl, P., Holík, L., Rogalewicz, A.: Forester: tool for verification of programs with pointers (2015). http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/
24. Tabakov, D., Vardi, M.: Model Checking Büchi Specifications. In LATA, volume Report 35/07. Research Group on Mathematical Linguistics, Universitat Rovira i Virgili, Tarragona (2007)

Efficient Inclusion Checking on Explicit and Semi-symbolic Tree Automata*

Lukáš Holík[1,2], Ondřej Lengál[1], Jiří Šimáček[1,3], and Tomáš Vojnar[1]

[1] FIT, Brno University of Technology, Czech Republic
[2] Uppsala University, Sweden
[3] VERIMAG, UJF/CNRS/INPG, Gières, France

Abstract. The paper considers several issues related to efficient use of tree automata in formal verification. First, a new efficient algorithm for inclusion checking on non-deterministic tree automata is proposed. The algorithm traverses the automaton downward, utilizing antichains and simulations to optimize its run. Results of a set of experiments are provided, showing that such an approach often very significantly outperforms the so far common upward inclusion checking. Next, a new semi-symbolic representation of non-deterministic tree automata, suitable for automata with huge alphabets, is proposed together with algorithms for upward as well as downward inclusion checking over this representation of tree automata. Results of a set of experiments comparing the performance of these algorithms are provided, again showing that the newly proposed downward inclusion is very often better than upward inclusion checking.

1 Introduction

Finite tree automata play a crucial role in several formal verification techniques, such as (abstract) regular tree model checking [3,5], verification of programs with complex dynamic data structures [6,11], analysis of network firewalls [7], and implementation of decision procedures of logics such as WS2S or MSO [15], which themselves have numerous applications (among the most recent and promising ones, let us mention at least verification of programs manipulating heap structures with data [16]).

Recently, there has been notable progress in the development of algorithms for efficient manipulation of non-deterministic finite tree automata (TA), more specifically, in solving the crucial problems of automata reduction [1] and of checking language inclusion [18,4,2]. As shown, e.g., in [4], replacing deterministic automata by non-deterministic ones can—in combination with the new methods for handling TA—lead to great efficiency gains. In this paper, we further advance the research on efficient algorithms for handling TA by (i) proposing a new algorithm for inclusion checking that turns out to significantly outperform the existing algorithms in most of our experiments and (ii) by presenting a semi-symbolic multi-terminal binary decision diagram (MTBDD) based representation of TA, together with various important algorithms for handling TA working over this representation.

* This work was supported by the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the BUT FIT project FIT-S-11-1, and the Swedish UPMARC project.

T. Bultan and P.-A. Hsiung (Eds.): ATVA 2011, LNCS 6996, pp. 243–258, 2011.

The classic textbook algorithm for checking inclusion $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ between two TA $\mathcal{A}_S$ (Small) and $\mathcal{A}_B$ (Big) first determinizes $\mathcal{A}_B$, computes the complement automaton $\overline{\mathcal{A}_B}$ of $\mathcal{A}_B$, and then checks language emptiness of the product automaton accepting $\mathcal{L}(\mathcal{A}_S) \cap \mathcal{L}(\overline{\mathcal{A}_B})$. This approach has been optimized in [18,4,2] which describe variants of this algorithm that try to avoid constructing the whole product automaton (which can be exponentially larger than $\mathcal{A}_B$ and which is indeed extremely large in many practical cases) by constructing its states and checking language emptiness on the fly. By employing the antichain principle [18,4], possibly combined with using upward simulation relations [2], the algorithm is often able to prove or refute inclusion by constructing a small part of the product automaton only[1]. We denote these algorithms as *upward* algorithms to reflect the direction in which they traverse automata $\mathcal{A}_S$ and $\mathcal{A}_B$.

The upward algorithms are sufficiently efficient in many practical cases. However, they have two drawbacks: (i) When generating the bottom-up post-image of a set $\mathcal{S}$ of sets of states, all possible n-tuples of states from all possible products $S_1 \times \ldots \times S_n$, $S_i \in \mathcal{S}$ need to be enumerated. (ii) Moreover, these algorithms are known to be compatible with only upward simulations as a means of their possible optimization, which is a disadvantage since downward simulations are often richer and also cheaper to compute.

The alternative *downward* approach to checking TA language inclusion was first proposed in [13] in the context of subtyping of XML types. This algorithm is not derivable from the textbook approach and has a more complex structure with its own weak points; nevertheless, it does not suffer from the two issues of the upward algorithm mentioned above. We generalize the algorithm of [13] for automata over alphabets with an arbitrary rank ([13] considers rank at most two), and, most importantly, we improve it significantly by using the antichain principle, empowered by a use of the cheap and usually large downward simulation. In this way, we obtain an algorithm which is complementary to and highly competitive with the upward algorithm as shown by our experimental results (in which the newly proposed algorithm significantly dominates in most of the considered cases).

Certain important applications of TA such as formal verification of programs with complex dynamic data structures or decision procedures of logics such as WS2S or MSO require handling very large alphabets. Here, the common choice is to use the MONA tree automata library [15] which is based on representing transitions of TA symbolically using MTBDDs. However, the encoding used by MONA is restricted to *deterministic* automata only. This implies a necessity of immediate determinisation after each operation over TA that introduces nondeterminism, which very easily leads to a state space explosion. Despite the extensive engineering effort spent to optimize the implementation of MONA, this fact significantly limits its applicability.

As a way to overcome this difficulty, we propose a semi-symbolic representation of *non-deterministic* TA which generalises the one used by MONA, and we develop

[1] The work of [18] does, in fact, not use the terminology of antichains despite implementing them in a symbolic, BDD-based way. It specialises to binary tree automata only. A more general introduction of antichains within a lattice-theoretic framework appeared in the context of word automata in [19]. Subsequently, [4] has generalized [19] for explicit upward inclusion checking on TA and experimentally advocated its use within abstract regular tree model checking [4]. See also [10] for other combinations of antichains and simulations for word automata.

algorithms implementing the basic operations on TA (such as union, intersection, etc.) as well as more involved algorithms for computing simulations and for checking inclusion (using simulations and antichains to optimize it) over the proposed representation. We also report on experiments with a prototype implementation of our algorithms showing again a dominance of downward inclusion checking and justifying usefulness of our symbolic encoding for TA with large alphabets.

The rest of this paper is organised as follows. Section 2 contains basic definitions for tree automata, tree automata languages, and simulations. Section 3 describes our downward inclusion checking algorithm and its experimental comparison with the upward algorithms. Further, Section 4 presents our MTBDD-based TA encoding, the algorithms working over this encoding, and an experimental evaluation of these algorithms. Section 5 then concludes the paper.

2 Preliminaries

A *ranked alphabet* Σ is a set of symbols together with a ranking function $\# : \Sigma \to \mathbb{N}$. For $a \in \Sigma$, the value $\#a$ is called the *rank* of a. For any $n \geq 0$, we denote by Σ_n the set of all symbols of rank n from Σ. Let ε denote the empty sequence. A *tree* t over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ that satisfies the following conditions: (1) $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$ and (2) for each $v \in dom(t)$, if $\#t(v) = n \geq 0$, then $\{i \mid vi \in dom(t)\} = \{1,\ldots,n\}$. Each sequence $v \in dom(t)$ is called a *node* of t. For a node v, we define the i^{th} *child* of v to be the node vi, and the i^{th} *subtree* of v to be the tree t' such that $t'(v') = t(viv')$ for all $v' \in \mathbb{N}^*$. A *leaf* of t is a node v which does not have any children, i.e., there is no $i \in \mathbb{N}$ with $vi \in dom(t)$. We denote by T_Σ the set of all trees over the alphabet Σ.

A (finite, non-deterministic) *tree automaton* (abbreviated sometimes as TA in the following) is a quadruple $\mathcal{A} = (Q, \Sigma, \Delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is a ranked alphabet, and Δ is a set of transition rules. Each transition rule is a triple of the form $((q_1,\ldots,q_n),a,q)$ where $q_1,\ldots,q_n,q \in Q, a \in \Sigma$, and $\#a = n$. We use equivalently $(q_1,\ldots,q_n) \xrightarrow{a} q$ and $q \xrightarrow{a} (q_1,\ldots,q_n)$ to denote that $((q_1,\ldots,q_n),a,q) \in \Delta$. The two notations correspond to the bottom-up and top-down representation of tree automata, respectively. (Note that we can afford to work interchangeably with both of them since we work with non-deterministic tree automata, which are known to have an equal expressive power in their bottom-up and top-down representations.) In the special case when $n = 0$, we speak about the so-called *leaf rules*, which we sometimes abbreviate as $\xrightarrow{a} q$ or $q \xrightarrow{a}$.

For an automaton $\mathcal{A} = (Q, \Sigma, \Delta, F)$, we use $Q^\#$ to denote the set of all tuples of states from Q with up to the maximum arity that some symbol in Σ has, i.e., if $r = max_{a \in \Sigma} \#a$, then $Q^\# = \bigcup_{0 \leq i \leq r} Q^i$. For $p \in Q$ and $a \in \Sigma$, we use $down_a(p)$ to denote the set of tuples accessible from p over a in the top-down manner; formally, $down_a(p) = \{(p_1,\ldots,p_n) \mid p \xrightarrow{a} (p_1,\ldots,p_n)\}$. For $a \in \Sigma$ and $(p_1,\ldots,p_n) \in Q^{\#a}$, we denote by $up_a((p_1,\ldots,p_n))$ the set of states accessible from $(p_1,\ldots,p_n)$ over the symbol a in the bottom-up manner; formally, $up_a((p_1,\ldots,p_n)) = \{p \mid (p_1,\ldots,p_n) \xrightarrow{a} p\}$. We also extend these notions to sets in the usual way, i.e., for $a \in \Sigma$, $P \subseteq Q$, and $R \subseteq Q^{\#a}$, $down_a(P) = \bigcup_{p \in P} down_a(p)$ and $up_a(R) = \bigcup_{(p_1,\ldots,p_n) \in R} up_a((p_1,\ldots,p_n))$.

Let $\mathcal{A} = (Q, \Sigma, \Delta, F)$ be a TA. A *run* of $\mathcal{A}$ over a tree $t \in T_\Sigma$ is a mapping $\pi : dom(t) \to Q$ such that, for each node $v \in dom(t)$ of rank $\#t(v) = n$ where $q = \pi(v)$, if $q_i = \pi(vi)$ for $1 \leq i \leq n$, then Δ has a rule $(q_1, \ldots, q_n) \xrightarrow{t(v)} q$. We write $t \xRightarrow{\pi} q$ to denote that π is a run of $\mathcal{A}$ over t such that $\pi(\varepsilon) = q$. We use $t \Longrightarrow q$ to denote that $t \xRightarrow{\pi} q$ for some run π. The *language* accepted by a state q is defined by $\mathcal{L}_\mathcal{A}(q) = \{t \mid t \Longrightarrow q\}$, while the language of a set of states $S \subseteq Q$ is defined as $\mathcal{L}_\mathcal{A}(S) = \bigcup_{q \in S} \mathcal{L}_\mathcal{A}(q)$. When it is clear which TA $\mathcal{A}$ we refer to, we only write $\mathcal{L}(q)$ or $\mathcal{L}(S)$. The language of $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_\mathcal{A}(F)$. We also extend the notion of a language to a tuple of states $(q_1, \ldots, q_n) \in Q^n$ by letting $\mathcal{L}((q_1, \ldots, q_n)) = \mathcal{L}(q_1) \times \cdots \times \mathcal{L}(q_n)$. The language of a set of n-tuples of sets of states $S \subseteq (2^Q)^n$ is the union of languages of elements of S, the set $\mathcal{L}(S) = \bigcup_{E \in S} \mathcal{L}(E)$. We say that X accepts y to express that $y \in \mathcal{L}(X)$.

A *downward simulation* on TA $\mathcal{A} = (Q, \Sigma, \Delta, F)$ is a preorder relation $\preceq_D \subseteq Q \times Q$ such that if $q \preceq_D p$ and $(q_1, \ldots, q_n) \xrightarrow{a} q$, then there are states $p_1, \ldots, p_n$ such that $(p_1, \ldots, p_n) \xrightarrow{a} p$ and $q_i \preceq_D p_i$ for each $1 \leq i \leq n$. Given a TA $\mathcal{A} = (Q, \Sigma, \Delta, F)$ and a downward simulation $\preceq_D$, an *upward simulation* $\preceq_U \subseteq Q \times Q$ induced by $\preceq_D$ is a relation such that if $q \preceq_U p$ and $(q_1, \ldots, q_n) \xrightarrow{a} q'$ with $q_i = q$, $1 \leq i \leq n$, then there are states $p_1, \ldots, p_n, p'$ such that $(p_1, \ldots, p_n) \xrightarrow{a} p'$ where $p_i = p$, $q' \preceq_U p'$, and $q_j \preceq_D p_j$ for each j such that $1 \leq j \neq i \leq n$.

3 Downward Inclusion Checking

Let us fix two tree automata $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$ for which we want to check whether $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ holds. If we try to answer this query top-down and we proceed in a naïve way, we immediately realize that the fact that the top-down successors of particular states are *tuples* of states leads us to checking inclusion of the languages of tuples of states. Subsequently, the need to compare the languages of each corresponding pair of states in these tuples will again lead to comparing the languages of tuples of states, and hence, we end up comparing the languages of *tuples of tuples* of states, and the need to deal with more and more nested tuples of states never stops.

For instance, given a transition $q \xrightarrow{a} (p_1, p_2)$ in $\mathcal{A}_S$, transitions $r \xrightarrow{a} (s_1, s_2)$ and $r \xrightarrow{a} (t_1, t_2)$ in $\mathcal{A}_B$, and assuming that there are no further top-down transitions from q and r, it holds that $\mathcal{L}(q) \subseteq \mathcal{L}(r)$ if and only if $\mathcal{L}((p_1, p_2)) \subseteq \mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2))$. Note that the union $\mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2))$ cannot be computed component-wise, this is, $\mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2)) \neq (\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times (\mathcal{L}(s_2) \cup \mathcal{L}(t_2))$. For instance, provided $\mathcal{L}(s_1) = \mathcal{L}(s_2) = \{b\}$ and $\mathcal{L}(t_1) = \mathcal{L}(t_2) = \{c\}$, it holds that $\mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2)) = \{(b,b), (c,c)\}$, but the component-wise union is $(\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times (\mathcal{L}(s_2) \cup \mathcal{L}(t_2)) = \{(b,b), (b,c), (c,b), (c,c)\}$. Hence, we cannot simply check whether $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1)$ and $\mathcal{L}(p_2) \subseteq \mathcal{L}(s_2) \cup \mathcal{L}(t_2)$ to answer the original query, and we have to proceed by checking inclusion on the obtained tuples of states. However, exploring the top-down transitions that lead from the states that appear in these tuples will lead us to dealing with tuples of tuples of states, etc.

Fortunately, there is a way out of the above trap. In particular, as first observed in [13] in the context of XML type checking, we can exploit the following property of the Cartesian product of sets $G, H \subseteq \mathcal{U}$: $G \times H = (G \times \mathcal{U}) \cap (\mathcal{U} \times H)$.

Hence, when we continue with our example, we get $\mathcal{L}((p_1,p_2)) = \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq \mathcal{L}((s_1,s_2)) \cup \mathcal{L}((t_1,t_2)) = (\mathcal{L}(s_1) \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times \mathcal{L}(t_2)) = ((\mathcal{L}(s_1) \times T_\Sigma) \cap (T_\Sigma \times \mathcal{L}(s_2))) \cup ((\mathcal{L}(t_1) \times T_\Sigma) \cap (T_\Sigma \times \mathcal{L}(t_2)))$. This can further be rewritten, using the distributive laws in the $(2^{T_\Sigma \times T_\Sigma}, \subseteq)$ lattice, as $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \cap ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2))) \cap ((T_\Sigma \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \cap ((T_\Sigma \times \mathcal{L}(s_2)) \cup (T_\Sigma \times \mathcal{L}(t_2)))$. It is easy to see that the inclusion holds exactly if it holds for all components of the intersection, i.e., if and only if $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \wedge \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2))) \wedge \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((T_\Sigma \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \wedge \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((T_\Sigma \times \mathcal{L}(s_2)) \cup (T_\Sigma \times \mathcal{L}(t_2)))$.

Two things should be noted in the above condition: (1) If we are computing the union of languages of two tuples such that they have T_Σ at all indices other than some index i, we can compute it component-wise. For instance, $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (\mathcal{L}(t_1) \times T_\Sigma)) = (\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times T_\Sigma$. This clearly holds iff $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1)$. (2) If T_Σ does not appear at the same positions as in the inclusion $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2)))$, it must hold that either $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1)$ or $\mathcal{L}(p_2) \subseteq \mathcal{L}(t_2)$.

Using the above observations, we can finally rewrite the equation $\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq \mathcal{L}((s_1,s_2)) \cup \mathcal{L}((t_1,t_2))$ into the following formula that does not contain languages of tuples but of single states only: $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1) \wedge (\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \vee \mathcal{L}(p_2) \subseteq \mathcal{L}(t_2)) \wedge (\mathcal{L}(p_1) \subseteq \mathcal{L}(t_1) \vee \mathcal{L}(p_2) \subseteq \mathcal{L}(s_2)) \wedge \mathcal{L}(p_2) \subseteq \mathcal{L}(s_2) \cup \mathcal{L}(t_2)$.

The above reasoning can be generalized to dealing with transitions of any arity as shown in Theorem 1, proved in [12]. In the theorem, we conveniently exploit the notion of *choice functions*. Given $P_B \subseteq Q_B$ and $a \in \Sigma$, $\#a = n \geq 1$, we denote by $cf(P_B,a)$ the set of all choice functions f that assign an index i, $1 \leq i \leq n$, to all n-tuples $(q_1,\ldots,q_n) \in Q_B^n$ such that there exists a state in P_B that can make a transition over a to $(q_1,\ldots,q_n)$; formally, $cf(P_B,a) = \{f : down_a(P_B) \to \{1,\ldots,\#a\}\}$.

Theorem 1. *Let $\mathcal{A}_S = (Q_S,\Sigma,\Delta_S,F_S)$ and $\mathcal{A}_B = (Q_B,\Sigma,\Delta_B,F_B)$ be tree automata. For sets $P_S \subseteq Q_S$ and $P_B \subseteq Q_B$ it holds that $\mathcal{L}(P_S) \subseteq \mathcal{L}(P_B)$ if and only if $\forall p_S \in P_S\ \forall a \in \Sigma$: if $p_S \xrightarrow{a} (r_1,\ldots,r_{\#a})$,*

$$\textit{then} \quad \begin{cases} down_a(P_B) = \{()\} & \textit{if } \#a = 0, \\ \forall f \in cf(P_B,a)\ \exists 1 \leq i \leq \#a : \mathcal{L}(r_i) \subseteq \bigcup\limits_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u}) = i}} \mathcal{L}(u_i) & \textit{if } \#a > 0. \end{cases}$$

3.1 Basic Algorithm of Downward Inclusion Checking

Next, we construct a basic algorithm for downward inclusion checking on tree automata $\mathcal{A}_S = (Q_S,\Sigma,\Delta_S,F_S)$ and $\mathcal{A}_B = (Q_B,\Sigma,\Delta_B,F_B)$. The algorithm is shown as Algorithm 1. Its main idea relies on a recursive application of Theorem 1 in function `expand1`. The function is given a pair $(p_S,P_B) \in Q_S \times 2^{Q_B}$ for which we want to prove that $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$—initially, the function is called for every pair (q_S,F_B) where $q_S \in F_S$. The function enumerates all possible top-down transitions that $\mathcal{A}_S$ can do from p_S (lines 3–8). For each such transition, the function either checks whether there is some transition $p_B \xrightarrow{a}$ for $p_B \in P_B$ if $\#a = 0$ (line 5), or it starts enumerating and recursively

checking queries $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ on which the result of $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ depends according to Theorem 1 (lines 9–16).

The `expand1` function keeps track of which inclusion queries are currently being evaluated in the set *workset* (line 2). Encountering a query $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ with $(p'_S, P'_B) \in \mathit{workset}$ means that the result of $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ depends on the result of $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ itself. In this case, the function immediately successfully returns because the result of the query then depends only on the other branches of the call tree.

Algorithm 1. Downward inclusion

Input: Tree automata $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S), \mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$
Output: *true* if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$, *false* otherwise

```
1  foreach q_S ∈ F_S do
2      if ¬expand1(q_S, F_B, ∅) then return false;
3  return true;
```

Function. `expand1` (p_S, P_B, *workset*)

```
   /* p_S ∈ Q_S,  P_B ⊆ Q_B,  and  workset ⊆ Q_S × 2^{Q_B}  */
1  if (p_S, P_B) ∈ workset then return true;
2  workset := workset ∪ {(p_S, P_B)};
3  foreach a ∈ Σ do
4      if #a = 0 then
5          if down_a(p_S) ≠ ∅ ∧ down_a(P_B) = ∅ then return false;
6      else
7          W := down_a(P_B);
8          foreach (r_1, ..., r_{#a}) ∈ down_a(p_S) do        /* p_S -a-> (r_1, ..., r_{#a})  */
9              foreach f ∈ {W → {1, ..., #a}} do
10                 found := false;
11                 foreach 1 ≤ i ≤ #a do
12                     S := {q_i | (q_1, ..., q_{#a}) ∈ W, f((q_1, ..., q_{#a})) = i};
13                     if expand1(r_i, S, workset) then
14                         found := true;
15                         break;
16                 if ¬found then return false;
17 return true;
```

Using Theorem 1 and noting that Algorithm 1 necessarily terminates because all its loops are bounded, and the recursion in function `expand1` is also bounded due to the use of *workset*, it is not difficult to see that the following theorem holds.

Theorem 2. *When applied on TA $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$, Algorithm 1 terminates and returns true if and only if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$.*

3.2 Optimized Algorithm of Downward Inclusion Checking

In this section, we propose several optimizations of the basic algorithm presented above that, according to our experiments, often have a huge impact on the efficiency of the algorithm—making it in many cases the most efficient algorithm for checking inclusion on tree automata that we are currently aware of. In general, the optimizations are based

Algorithm 2. Downward inclusion (antichains + preorder)

Input: TA $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S), \mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B), \preceq \subseteq (Q_S \cup Q_B)^2$
Output: *true* if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$, *false* otherwise
Data: $NN := \emptyset$
foreach $q_S \in F_S$ **do**
  if $\neg$`expand2`$(q_S, F_B, \emptyset)$ **then return** *false*;
return *true*;

Function. `expand2` (p_S, P_B, *workset*)

/* $p_S \in Q_S$, $P_B \subseteq Q_B$, and *workset* $\subseteq Q_S \times 2^{Q_B}$ */
if $\exists (p'_S, P'_B) \in workset : p_S \preceq p'_S \wedge P'_B \preceq^{\forall\exists} P_B$ **then return** *true*;
if $\exists (p'_S, P'_B) \in NN : p'_S \preceq p_S \wedge P_B \preceq^{\forall\exists} P'_B$ **then return** *false* ;
if $\exists p \in P_B : p_S \preceq p$ **then return** *true*;
$workset := workset \cup \{(p_S, P_B)\}$;
foreach $a \in \Sigma$ **do**
  if $\#a = 0$ **then**
    if $down_a(p_S) \neq \emptyset \wedge down_a(P_B) = \emptyset$ **then return** *false*;
  else
    $W := down_a(P_B)$;
    foreach $(r_1, \ldots, r_{\#a}) \in down_a(p_S)$ **do** /* $p_S \xrightarrow{a} (r_1, \ldots, r_{\#a})$ */
      foreach $f \in \{W \to \{1, \ldots, \#a\}\}$ **do**
        $found := false$;
        foreach $1 \leq i \leq \#a$ **do**
          $S := \{q_i \mid (q_1, \ldots, q_{\#a}) \in W, f((q_1, \ldots, q_{\#a})) = i\}$;
          if `expand2`$(r_i, S, workset)$ **then**
            $found := true$;
            break;
          if $\nexists (r', H) \in NN : r' \preceq r_i \wedge S \preceq^{\forall\exists} H$ **then**
            $NN := (NN \setminus \{(r', H) \mid H \preceq^{\forall\exists} S, r_i \preceq r'\}) \cup \{(r_i, S)\}$;
        if $\neg found$ **then return** *false*;
return *true*;

on an original use of simulations and antichains in a way suitable for the context of downward inclusion checking.

In what follows, we assume that there is available a preorder $\preceq \subseteq (Q_S \cup Q_B)^2$ compatible with language inclusion, i.e., such that $p \preceq q \implies \mathcal{L}(p) \subseteq \mathcal{L}(q)$, and we use $P \preceq^{\forall\exists} R$ where $P, R \subseteq (Q_S \cup Q_B)^2$ to denote that $\forall p \in P \, \exists r \in R : p \preceq r$. An example of such a preorder, which can be efficiently computed, is the (maximal) downward simulation $\preceq_D$. We propose the following concrete optimizations of the downward checking of $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$:

1. If there exists a state $p_B \in P_B$ such that $p_S \preceq p_B$, then the inclusion clearly holds (from the assumption made about $\preceq$), and no further checking is needed.
2. Next, it can be seen without any further computation that the inclusion does *not* hold if there exists some (p'_S, P'_B) such that $p'_S \preceq p_S$ and $P_B \preceq^{\forall\exists} P'_B$, and we have already established that $\mathcal{L}(p'_S) \not\subseteq \mathcal{L}(P'_B)$. Indeed, we have $\mathcal{L}(P_B) \subseteq \mathcal{L}(P'_B) \not\supseteq \mathcal{L}(p'_S) \subseteq \mathcal{L}(p_S)$, and therefore $\mathcal{L}(p_S) \not\subseteq \mathcal{L}(P_B)$.
3. Finally, we can stop evaluating the given inclusion query if there is some $(p'_S, P'_B) \in$ *workset* such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$. Indeed, this means that the result of $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ depends on the result of $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$. However, if $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ holds, then also $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ holds because we have $\mathcal{L}(p_S) \subseteq \mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B) \subseteq \mathcal{L}(P_B)$.

Table 1. Percentages of cases in which the respective methods were the fastest

Size	Pairs	Timeout	Up	Up+s	Down	Down+s	Avg up speedup	Avg down speedup
50–250	323	20 s	31.21 %	0.00 %	53.50 %	15.29 %	1.71	3.55
400–600	64	60 s	9.38 %	0.00 %	39.06 %	51.56 %	0.34	46.56

The version of Algorithm 1 including all the above proposed optimizations is shown as Algorithm 2. The optimizations can be found in the function `expand2` that replaces the function `expand1`. In particular, line 3 implements the first optimization, line 2 the second one, and line 1 the third one. In order to implement the second optimization, the algorithm maintains a new set NN. This set stores pairs (p_S, P_B) for which it has already been shown that the inclusion $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ does *not* hold[2].

As a further optimization, the set NN is maintained as an antichain w.r.t. the preorder that compares the pairs stored in NN such that the states from Q_S on the left are compared w.r.t. $\preceq$, and the sets from 2^{Q_B} on the right are compared w.r.t. $\succeq^{\exists\forall}$ (line 19). Clearly, there is no need to store a pair (p_S, P_B) that is bigger in the described sense than some other pair (p'_S, P'_B) since every time (p_S, P_B) can be used to prune the search, (p'_S, P'_B) can also be used.

Taking into account Theorem 2 and the above presented facts, it is not difficult to see that the following holds.

Theorem 3. *When applied on TA $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$, Algorithm 2 terminates and returns true if and only if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$.*

3.3 Experimental Results

We have implemented Algorithm 1 (which we mark as `Down` in what follows) as well as Algorithm 2 using the maximum downward simulation as the input preorder (which is marked as `Down+s` below). We have also implemented the algorithm of upward inclusion checking using antichains from [4] and its modification using upward simulation proposed in [2] (these algorithms are marked as `Up` and `Up+s` below). We tested our approach on 387 tree automata pairs of different sizes generated from the intermediate steps of abstract regular tree model checking of the algorithm for rebalancing red-black trees after insertion or deletion of a leaf node [4].

The results of the experiments are presented in the following tables. Table 1 compares the methods according to the percentage of the cases in which they were the fastest when checking inclusion on the same automata pair. The results are grouped into two sets according to the size of the automata measured in the number of states. The table also gives the average speedup of the fastest upward approach compared to the fastest downward approach in case the upward computation was faster than the downward one (and vice versa). Table 2 provides a comparison of the methods that

[2] In [12], a further optimization exploiting that $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ has been shown to hold is proposed, but it is much more complicated in order to avoid memorizing possibly invalid assumptions made during the computation.

use simulation (either upward for `Up+s` or downward for `Down+s`) without counting the time for computing simulation (in such cases they were always faster than the methods not using simulations). This comparison is motivated by the observation that inclusion checking may be used as a part of a bigger computation that anyway computes the simulation relations (which happens, e.g., in abstract regular model checking where the simulations are used for reducing the size of the encountered automata). Finally, Table 3 summarizes how often the particular methods were successful in our testing runs (i.e., how often they did not timeout.).

Table 2. Percentages of cases in which the methods were the fastest when not counting the time for computing the simulation

Size	Pairs	Timeout	`Up+s`	`Down+s`	Avg up speedup	Avg down speedup
50–250	323	20 s	81.82 %	18.18 %	1.33	3.60
400–600	64	60 s	20.31 %	79.69 %	9.92	2116.29

Table 3. Percentages of successful runs that did not timeout

Size	Pairs	Timeout	`Up`	`Up+s`	`Down`	`Down+s`
50–250	323	20 s	100.00 %	100.00 %	74.92 %	99.07 %
400–600	64	60 s	51.56 %	51.56 %	39.06 %	90.62 %

The results show that the overhead of computing upward simulation is too high in all the cases that we have considered, causing upward inclusion checking using simulation to be the slowest when the time for computing the simulation used by the algorithm is included[3]. Next, it can be seen that for each of the remaining approaches there are cases in which they win in a significant way. However, the downward approaches are clearly dominating in significantly more of our test cases (with the only exception being the case of small automata when the time of computing simulations is not included). Moreover, the dominance of the downward checking increases with the size of the automata that we considered in our test cases.

4 Semi-symbolic Representation of Tree Automata

We next consider a natural, semi-symbolic, MTBDD-based encoding of non-deterministic TA, suitable for handling automata with huge alphabets. We propose algorithms for computing downward simulations and for efficient downward inclusion checking on the considered representation. Due to space restrictions, we defer algorithms for further operations on the considered semi-symbolic representation of TA, including upward inclusion checking, to [12].

4.1 Binary Decision Diagrams

Let $\mathbb{B} = \{0,1\}$ be the set of Boolean values. A *Boolean function* of *arity* k is a function of the form $f : \mathbb{B}^k \to \mathbb{B}$. We extend the notion of Boolean functions to an arbitrary

[3] Note that `Up+s` was winning over `Up` in the experiments of [2] even with the time for computing simulation included, which seems to be caused by a much less efficient implementation of the antichains in the original algorithm.

nonempty set S where a k-ary Boolean function extended to the domain set S is a function of the form $f : \mathbb{B}^k \to S$.

A *reduced ordered binary decision diagram* (ROBDD) [8] r over n Boolean variables $x_1, \ldots, x_n$ is a connected directed acyclic graph with a single *source node* (denoted as *r.root*) and at least one of the two *sink nodes* **0** and **1**. We call *internal* the nodes which are not sink nodes. A function *var* assigns each internal node a Boolean variable from the set $X = \{x_1, \ldots, x_n\}$, which is assumed to be ordered by the ordering $x_1 < x_2 < \cdots < x_n$. For every internal node v there exist 2 outgoing edges labelled *low* and *high*. We denote by *v.low* a node w and by *v.high* a node z such that there exists a directed edge from v to w labelled by *low* and a directed edge from v to z labelled by *high*, respectively. For each internal node v, it must hold that $var(v) < var(v.low)$ and $var(v) < var(v.high)$ and also $v.low \neq v.high$. A node v represents an n-ary Boolean function $[\![v]\!] : \mathbb{B}^n \to \mathbb{B}$ that assigns to each assignment to the Boolean variables in X a corresponding Boolean value defined in the following way (using $\overline{x}$ as an abbreviation for $x_1 \ldots x_n$): $[\![\mathbf{0}]\!] = \lambda\, \overline{x}\, .\, 0$, $[\![\mathbf{1}]\!] = \lambda\, \overline{x}\, .\, 1$, and $[\![v]\!] = \lambda\, \overline{x}\, .\, (\neg x_i \wedge [\![v.low]\!]) \vee (x_i \wedge [\![v.high]\!])$ for $var(v) = x_i$. For every two nodes v and w, it holds that $v \neq w \implies [\![v]\!] \neq [\![w]\!]$. We say that an ROBDD r represents the Boolean function $[\![r]\!] = [\![r.root]\!]$. Dually, for a Boolean function f, we use $\langle f \rangle$ to denote the ROBDD representing f, i.e., $f = [\![\langle f \rangle]\!]$.

We generalise the standard *Apply* operation for manipulation of Boolean functions represented by ROBDDs in the following way: let op_1, op_2, and op_3 be in turn arbitrary unary, binary, and ternary Boolean functions. Then the functions $Apply_1$, $Apply_2$, and $Apply_3$ produce a new ROBDD which is defined as follows for ROBDDs f, g, and h: $Apply_1(f, op_1) = \langle \lambda\, \overline{x}\, .\, op_1([\![f(\overline{x})]\!]) \rangle$, $Apply_2(f, g, op_2) = \langle \lambda\, \overline{x}\, .\, op_2([\![f(\overline{x})]\!], [\![g(\overline{x})]\!]) \rangle$, and $Apply_3(f, g, h, op_3) = \langle \lambda\, \overline{x}\, .\, op_3([\![f(\overline{x})]\!], [\![g(\overline{x})]\!], [\![h(\overline{x})]\!]) \rangle$. In practice, one can also use *Apply* operations with side-effects.

The notion of ROBDDs is further generalized to *multi-terminal binary decision diagrams* (MTBDDs) [9]. MTBDDs are essentially the same data structures as ROBDDs, the only difference being the fact that the set of sink nodes is not restricted to two nodes. Instead, it can contain an arbitrary number of nodes labelled uniquely by elements of an arbitrary domain set S. All standard notions for ROBDDs can naturally be extended to MTBDDs. An MTBDD m then represents a Boolean function extended to S, i.e., $[\![m]\!] : \mathbb{B}^n \to S$. Further, the concept of *shared MTBDDs* is used. A shared MTBDD s is an MTBDD with multiple source nodes (or *roots*) that represents a mapping of every element of the set of roots R to a function induced by the MTBDD corresponding to the the given root, i.e., $[\![s]\!] : R \to (\mathbb{B}^n \to S)$.

4.2 Encoding the Transition Function of a TA Using Shared MTBDDs

We fix a tree automaton $\mathcal{A} = (Q, \Sigma, \Delta, F)$ for the rest of the section. We consider both a top-down and a bottom-up representation of its transition function. This is because some operations on $\mathcal{A}$ are easier to do on the former representation while others on the latter. We assume w.l.o.g. that the input alphabet Σ of $\mathcal{A}$ is represented in binary using n bits. We assign each bit in the binary encoding of Σ a Boolean variable from the set $\{x_1, \ldots, x_n\}$. We can then use shared MTBDDs with a set of roots R and a domain set S for encoding the various functions of the form $R \to (\Sigma \to S)$ that we will need.

Our *bottom-up* representation of the transition function Δ of the TA $\mathcal{A}$ uses a shared MTBDD Δ^{bu} over Σ where the set of root nodes is $Q^{\#}$, and the domain of labels of

sink nodes is 2^Q. The MTBDD Δ^{bu} represents a function $[\![\Delta^{bu}]\!] : Q^\# \to (\Sigma \to 2^Q)$ defined as $[\![\Delta^{bu}]\!] = \lambda\ (q_1,\ldots,q_p)\ a\ .\ \{q \mid (q_1,\ldots,q_p) \xrightarrow{a} q\}$. It clearly holds that $[\![\Delta^{bu}((q_1,\ldots,q_p),a)]\!] = up_a((q_1,\ldots,q_p))$.

Our *top-down* representation of the transition function Δ of the TA $\mathcal{A}$ uses a shared MTBDD Δ^{td} over Σ where the set of root nodes is Q, and the domain of labels of sink nodes is $2^{Q^\#}$. The MTBDD Δ^{td} represents a function $[\![\Delta^{td}]\!] : Q \to (\Sigma \to 2^{Q^\#})$ defined as $[\![\Delta^{td}]\!] = \lambda\, q\, a\ .\ \{(q_1,\ldots,q_p) \mid q \xrightarrow{a} (q_1,\ldots,q_p)\}$. Clearly, $[\![\Delta^{td}(q,a)]\!] = down_a(q)$.

Sometimes it is necessary to convert between the bottom-up and top-down representation of a TA. For instance, when computing downward simulations (as explained below), one needs to switch between the bottom-up and top-down representation. Fortunately, the two representations are easy to convert (cf. [12]).

4.3 Downward Simulation on Semi-symbolically Encoded TA

We next give an algorithm for computing the maximum downward simulation relation on the states of the TA $\mathcal{A}$ whose transition function is encoded using our semi-symbolic representation. The algorithm is inspired by the algorithm from [14] proposed for computing simulations on finite (word) automata. For use in the algorithm, we extend the notion of downward simulation to tuples of states by defining $(q_1,\ldots,q_n) \preceq_D (r_1,\ldots,r_n)$ to hold iff $\forall 1 \leq i \leq n : q_i \preceq_D r_i$.

Our algorithm for computing downward simulations, shown as Algorithm 3, starts with a gross over-approximation of the maximum downward simulation, which is then pruned until the maximum downward simulation is obtained. The algorithm uses the following main data structures:

- For each $q \in Q$, $sim(q) \subseteq Q$ is the set of states that are considered to simulate q at the current step of the computation. Its value is gradually pruned during the computation. At the end, it encodes the maximum downward simulation being computed.
- The set $remove \subseteq Q^\# \times Q^\#$ contains pairs $((q_1,\ldots,q_n),(r_1,\ldots,r_n))$ of tuples of states, for which it is known that $(q_1,\ldots,q_n) \not\preceq_D (r_1,\ldots,r_n)$, for processing.
- Finally, cnt is a shared MTBDD encoding a function $[\![cnt]\!] : Q^\# \to (\Sigma \to (Q \to \mathbb{N}))$ that for each $(q_1,\ldots,q_n) \in Q^\#$, $a \in \Sigma$, and $q \in Q$, gives a value $h \in \mathbb{N}$ such that $(q_1,\ldots,q_n)$ can make a bottom-up transition over a to h distinct states $r \in sim(q)$.

The algorithm works in two phases. We assume that we start with a TA whose transition function is represented bottom-up. In the *initialization* phase, the dual top-down representation of the transition function is first computed (note that we can also start with a top-down representation and compute the bottom-up representation as both are needed in the algorithm). The three main data structures are then initialized as follows:

- For each $q \in Q$, the set $sim(q)$ is initialized as the set of states that can make top-down transitions over the same symbols as q, which is determined using the *Apply* operation on line 9. This is, when starting the main computation loop on line 17, the value of sim for each state $q \in Q$ is $sim(q) = \{r \mid \forall a \in \Sigma : q \xrightarrow{a} (q_1,\ldots,q_n) \implies r \xrightarrow{a} (r_1,\ldots,r_n)\}$.
- The *remove* set is initialized to contain each pair of tuples of states $((q_1,\ldots,q_n),(r_1,\ldots,r_n))$ for which it holds that the relation $(q_1,\ldots,q_n) \preceq_D (r_1,\ldots,r_n)$ is broken

Algorithm 3. Computing downward simulation on semi-symbolic TA

Input: Tree automaton $\mathcal{A} = (Q, \Sigma, \Delta^{bu}, F)$
Output: Maximum downward simulation $\preceq_D \subseteq Q^2$

```
   /* initialization */
1  Δ^td := invertMTBDD(Δ^bu);
2  remove := ∅;
3  initCnt := ⟨λ a . ∅⟩ ;                                   /* [[initCnt]] : Σ → (Q → ℕ) */
4  foreach q ∈ Q do
5      sim(q) := ∅;
6      initCnt := Apply2(Δ^td(q), initCnt, (λ X Y . Y ∪ {(q, |X|)}));
7      foreach r ∈ Q do
8          isSim := true;
9          Apply2(Δ^td(q), Δ^td(r), (λ X Y . if (X ≠ ∅ ∧ Y = ∅) then isSim := false)) ;
10         if isSim then
11             sim(q) := sim(q) ∪ {r};
12         else
13             foreach (q1,...,qn) ∈ Q^#, (r1,...,rn) ∈ Q^# : ∃1 ≤ i ≤ n : qi = q ∧ ri = r do
14                 remove := remove ∪ {((q1,...,qn),(r1,...,rn))};
15 cnt := ⟨λ (q1,...,qn) a . ∅⟩ ;                           /* [[cnt]] : Q^# → (Σ → (Q → ℕ)) */
16 foreach (q1,...,qn) ∈ Q^# do cnt((q1,...,qn)) := initCnt;
   /* computation */
17 while ∃((q1,...,qn),(r1,...,rn)) ∈ remove do
18     remove := remove \ {((q1,...,qn),(r1,...,rn))};
19     cnt((q1,...,qn)) :=
       Apply3(Δ^bu((r1,...,rn)), Δ^bu((q1,...,qn)), cnt((q1,...,qn)), (refine sim remove));
20 return {(q,r) | q ∈ Q, r ∈ sim(q)};
```

Function. `refine(&sim, &remove, up_aR, up_aQ, cnt_aQ)`

```
1  newCnt_aQ := cnt_aQ;
2  foreach s ∈ up_aR do
3      newCnt_aQ(s) := newCnt_aQ(s) − 1;
4      if newCnt_aQ(s) = 0 then
5          foreach p ∈ up_aQ : s ∈ sim(p) do
6              foreach (p1,...,pn) ∈ Q^#, (s1,...,sn) ∈ Q^# : ∃1 ≤ i ≤ n : pi = p ∧ si = s do
7                  if ∀1 ≤ j ≤ n : sj ∈ sim(pj) then
8                      remove := remove ∪ {((p1,...,pn),(s1,...,sn))};
9              sim(p) := sim(p) \ {s};
10 return newCnt_aQ;
```

even for the initial approximation of $\preceq_D$, i.e., for some position $1 \leq i \leq n$ there is a pair $q_i, r_i \in Q$ such that $r_i \notin sim(q_i)$.

– To initialize the shared MTBDD *cnt*, the algorithm constructs an auxiliary MTBDD *initCnt* representing a function $[\![initCnt]\!] : \Sigma \to (Q \to \mathbb{N})$. Via the *Apply* operation on line 6, this MTBDD gradually collects, for each symbol $a \in \Sigma$, the set of pairs (q,h) such that q can make a top-down transition to h distinct tuples over the symbol a. This MTBDD is then copied to the shared MTBDD *cnt* for each tuple of states $(q_1,\ldots,q_n) \in Q^{\#}$. This is justified by the fact that we start by assuming that the simulation relation is equal to $Q \times Q$, which for a symbol $a \in \Sigma$ and a pair $(q,h) \in cnt((q_1,\ldots,q_n))$ means that $(q_1,\ldots,q_n)$ can make a bottom-up transition over a to h distinct states $r \in sim(q)$.

The main *computation* phase gradually restricts the initial over-approximation of the maximum downward simulation being computed. As we have said, the *remove* set contains pairs $((q_1,\ldots,q_n),(r_1,\ldots,r_n))$ for which it holds that $(q_1,\ldots,q_n)$ cannot be simulated by $(r_1,\ldots,r_n)$, i.e., $(q_1,\ldots,q_n) \not\preceq_D (r_1,\ldots,r_n)$. When such a pair is processed, the algorithm decrements the counter $[\![cnt((q_1,\ldots,q_n),a,s)]\!]$ for each state s for which there exists a bottom-up transition over a symbol $a \in \Sigma$ such that $(r_1,\ldots,r_n) \xrightarrow{a} s$. The meaning is that s can make one less top-down transition over a to some $(t_1,\ldots,t_n)$ such that $(q_1,\ldots,q_n) \preceq_D (t_1,\ldots,t_n)$. If $[\![cnt((q_1,\ldots,q_n),a,s)]\!]$ drops to zero, it means that s cannot make a top-down transition over a to any $(t_1,\ldots,t_n)$ such that $(q_1,\ldots,q_n) \preceq_D (t_1,\ldots,t_n)$. This means, for all $p \in Q$ such that p can make a top-down transition over a to $(q_1,\ldots,q_n)$, that s no longer simulates p, i.e., $p \not\preceq_D s$. When the simulation relation between p and s, $p \preceq_D s$, is broken, then the simulation relation between all m-tuples $(p_1,\ldots,p_m)$ and $(s_1,\ldots,s_m)$ such that $\exists 1 \leq j \leq m : p_j = p \wedge s_j = s$ must also be broken, therefore the pair $((p_1,\ldots,p_m),(s_1,\ldots,s_m))$ is put to the *remove* set (unless the simulation relation between some other states in the tuples has already been broken before).

Correctness of the algorithm is summarised in the below theorem, which can be proven analogically as correctness of the algorithm proposed in [14], taking into account the meaning of the above described MTBDD-based structures and the operations performed on them.

Theorem 4. *When applied on a TA $\mathcal{A} = (Q,\Sigma,\Delta,F)$ whose transition function is encoded semi-symbolically in the bottom-up way as Δ^{bu}, Algorithm 3 terminates and returns the maximum downward simulation on Q.*

4.4 Downward Inclusion Checking on Semi-symbolically Encoded TA

We now proceed to an algorithm of efficient downward inclusion checking on semi-symbolically represented TA. The algorithm we propose for this purpose is derived from Algorithm 2 by plugging the `expand3` function instead of the `expand2` function. It is based on the same basic principle as `expand2`, but it has to cope with the symbolically encoded transition relation. In particular, in order to inspect whether for a pair (p_S,P_B) and all symbols $a \in \Sigma$ the inclusion between each tuple from $down_a(p_S)$ and the set of tuples $down_a(P_B)$ holds, the *doesInclusionHold* parameter initialized to *true* is passed to the *Apply* operation on line 9 of the `expand3` function. If the algorithm finds out that the inclusion does not hold in some execution of the `procDown` function in the context of a single *Apply*, *doesInclusionHold* is assigned the *false* value, which is later returned by `expand3`. Otherwise `expand3` returns its original *true* value.

4.5 Experimental Results

We have implemented a prototype of a library for working with TA encoded semi-symbolically as described above. We have used the CUDD library [17] as an implementation of shared MTBDDs. The prototype contains the algorithms presented in this section and some more presented in [12]. The results on downward inclusion checking that we have obtained with the explicitly represented TA encouraged us to also compare performance of the upward inclusion checking and downward inclusion checking on automata with large alphabets using our prototype.

Function. `expand3` (p_S, P_B, *workset*)

```
   /* p_S ∈ Q_S , P_B ⊆ Q_B , and workset ⊆ Q_S × 2^{Q_B} */
 1 if ∃(p'_S, P'_B) ∈ workset : p_S ⪯ p'_S ∧ P'_B ⪯^{∀∃} P_B then return true;
 2 if ∃(p'_S, P'_B) ∈ NN : p'_S ⪯ p_S ∧ P_B ⪯^{∀∃} P'_B then return false ;
 3 if ∃p ∈ P_B : p_S ⪯ p then return true;
 4 workset := workset ∪ {(p_S, P_B)};
 5 tmp := ⟨λ a . ∅⟩;
 6 foreach p_B ∈ P_B do
 7     tmp := Apply_2(tmp, Δ^td_B(p_B), (λ X Y . X ∪ Y));
 8 doesInclusionHold := true;
 9 Apply_2(Δ^td_S(p_S), tmp, (procDown doesInclusionHold workset));
10 return doesInclusionHold;
```

Function. `procDown` (*&doesInclusionHold, &workset, down$_a$p$_S$, down$_a$P$_B$*)

```
 1 if () ∈ down_a p_S ∧ () ∉ down_a P_B then
 2     doesInclusionHold := false;
 3 else
 4     W := down_a P_B;
 5     foreach (r_1, ..., r_n) ∈ down_a p_S do          /* p_S -a-> (r_1, ..., r_n) */
 6         foreach f ∈ {W → {1, ..., n}} do
 7             found := false;
 8             foreach 1 ≤ i ≤ n do
 9                 S := {q_i | (q_1, ..., q_n) ∈ W, f((q_1, ..., q_n)) = i};
10                 if expand3(r_i, S, workset) then
11                     found := true;
12                     break;
13                 if ∄(r', H) ∈ NN : r' ⪯ r_i ∧ S ⪯^{∀∃} H then
14                     NN := (NN \ {(r', H) | H ⪯^{∀∃} S, r_i ⪯ r'}) ∪ {(r_i, S)};
15             if ¬found then
16                 doesInclusionHold := false;
17                 return;
```

We have compared the upward inclusion checking algorithm from [4] adapted for semi-symbolically represented tree automata, which is given in [12] (and marked as `UpSym` in the following), with the downward inclusion checking algorithm presented above. In the latter case, we let the algorithm use either the identity relation, which corresponds to downward inclusion checking without using any simulation (this case is marked as `DownSym` below), or the maximum downward simulation (which is marked as `DownSym+s` in the results). We have not considered upward inclusion checking with upward simulation due to its negative results in our experiments with explicitly encoded automata[4]. For the comparison, we used 97 pairs of tree automata with a large alphabet which we encoded into 12 bits. The size of the automata was between 50 and 150 states and the timeout was set to 300 s. The automata were obtained by taking the automata considered in Section 3.3 and labelling their transitions by randomly generated sets of symbols from the considered large alphabet.

The results that we have obtained are presented in the following tables. Table 4 compares the methods according to the percentage of the cases in which they were the

[4] We, however, note that possibilities of implementing upward inclusion checking combined with upward simulations over semi-symbolically encoded TA and a further evaluation of this algorithm are still interesting subjects for the future.

fastest when checking inclusion on the same automata pair. This table also presents the average speedup of the upward approach compared to the fastest downward approach in case the upward computation was faster than the downward one (and vice versa). Table 5 summarizes how often each of the methods was successful in the testing runs.

When we compare the above experimental results with the results obtained on the explicitly represented automata presented in Section 3.3, we may note that (1) downward inclusion checking is again significantly dominating, but (2) the advantage of exploiting downward simulation has decreased. According to the information we gathered from code profiling of our implementation, this is due to the overhead of the CUDD library which is used as the underlying layer for storage of shared MTBDDs of several data structures (which indicates a need of a different MTBDD library to be used or perhaps of a specialised MTBDD library to be developed).

Table 4. Percentages of cases in which the respective methods were the fastest

UpSym	DownSym	DownSym+s	Avg up speedup	Avg down speedup
6.67 %	90.67 %	2.67 %	24.39	4389.76

Table 5. Successful runs that did not timeout (in %)

UpSym	DownSym	DownSym+s
77.32 %	77.32 %	26.08 %

We also evaluated performance of the implementation of the described algorithms using a semi-symbolic encoding of TA with performance of the algorithms using an explicit encoding of TA considered in Section 3 on the automata with the large alphabet. The symbolic version was in average 8676 times faster than the explicit one as expected when using a large alphabet.

5 Conclusion

We have proposed a new algorithm for checking language inclusion over non-deterministic TA (based on the one from [13]) that traverses automata in the downward manner and uses both antichains and simulations to optimize its computation. This algorithm is, according to our experimental results, mostly superior to the known upward algorithms. We have further presented a semi symbolic MTBDD-based representation of non-deterministic TA generalising the one used by MONA, together with important tree automata algorithms working over this representation, most notably an algorithm for computing downward simulations over TA inspired by [14] and the downward language inclusion algorithm improved by simulations and antichains proposed in this paper. We have experimentally justified usefulness of the symbolic encoding for non-deterministic TA with large alphabets.

Our experimental results suggest that the MTBDD package CUDD is not very efficient for our purposes and that better results could probably be achieved using a specialised MTBDD package whose design is an interesting subject for further work. Apart from that, it would be interesting to encode antichains used within the language inclusion checking algorithms symbolically as, e.g., in [18]. An interesting problem here is how to efficiently encode antichains based not on the subset inclusion but on a

simulation relation. Finally, as a general target, we plan to continue in our work towards obtaining a really efficient TA library which could ultimately replace the one of MONA.

References

1. Abdulla, P.A., Bouajjani, A., Holík, L., Kaati, L., Vojnar, T.: Computing Simulations over Tree Automata: Efficient Techniques for Reducing Tree Automata. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 93–108. Springer, Heidelberg (2008)
2. Abdulla, P.A., Holík, L., Chen, Y.-F., Mayr, R., Vojnar, T.: When Simulation Meets Antichains (On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata). In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
3. Abdulla, P.A., Jonsson, B., Mahata, P., d'Orso, J.: Regular Tree Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 555–568. Springer, Heidelberg (2002)
4. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008)
5. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking. ENTCS, vol. 149. Elsevier, Amsterdam (2006)
6. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
7. Bourdier, T.: Tree Automata-based Semantics of Firewalls. In: Proc. of SAR-SSI 2011. IEEE, Los Alamitos (2011)
8. Bryant, R.E.: Graph-based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers (1986)
9. Clarke, E.M., McMillan, K.L., Zhao, X., Fujita, M., Yang, J.: Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping. FMSD 10 (1997)
10. Doyen, L., Raskin, J.F.: Antichain Algorithms for Finite Automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 2–22. Springer, Heidelberg (2010)
11. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: orest Automata for Verification of Heap Manipulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 424–440. Springer, Heidelberg (2011)
12. Holík, L., Lengál, O., Šimáček, J., Vojnar, T.: Efficient Inclusion Checking on Explicit and Semi-Symbolic Tree Automata. Tech. rep. FIT-TR-2011-04, FIT BUT, Czech Rep. (2011)
13. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular Expression Types for XML. ACM Trans. Program. Lang. Syst. 27 (2005)
14. Ilie, L., Navarro, G., Yu, S.: On NFA Reductions. In: Karhumäki, J., Maurer, H., Păun, G., Rozenberg, G. (eds.) Theory Is Forever. LNCS, vol. 3113, pp. 112–124. Springer, Heidelberg (2004)
15. Klarlund, N., Møller, A., Schwartzbach, M.I.: MONA Implementation Secrets. International Journal of Foundations of Computer Science, 13(4) (2002)
16. Madhusudan, P., Parlato, G., Qiu, X.: Decidable Logics Combining Heap Structures and Data. SIGPLAN Not. 46 (2011)
17. Somenzi, F.: CUDD: CU Decision Diagram Package Release 2.4.2 (May 2011)
18. Tozawa, A., Hagiya, M.: XML Schema Containment Checking Based on Semi-implicit Techniques. In: Ibarra, O.H., Dang, Z. (eds.) CIAA 2003. LNCS, vol. 2759, pp. 213–225. Springer, Heidelberg (2003)
19. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)

String Constraints with Concatenation and Transducers Solved Efficiently

LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic
PETR JANKŮ, Brno University of Technology, Czech Republic
ANTHONY W. LIN, University of Oxford, United Kingdom
PHILIPP RÜMMER, Uppsala University, Sweden
TOMÁŠ VOJNAR, Brno University of Technology, Czech Republic

String analysis is the problem of reasoning about how strings are manipulated by a program. It has numerous applications including automatic detection of cross-site scripting, and automatic test-case generation. A popular string analysis technique includes symbolic executions, which at their core use constraint solvers over the string domain, a.k.a. string solvers. Such solvers typically reason about constraints expressed in theories over strings with the concatenation operator as an atomic constraint. In recent years, researchers started to recognise the importance of incorporating the replace-all operator (i.e. replace all occurrences of a string by another string) and, more generally, finite-state transductions in the theories of strings with concatenation. Such string operations are typically crucial for reasoning about XSS vulnerabilities in web applications, especially for modelling sanitisation functions and implicit browser transductions (e.g. innerHTML). Although this results in an undecidable theory in general, it was recently shown that the straight-line fragment of the theory is decidable, and is sufficiently expressive in practice. In this paper, we provide the first string solver that can reason about constraints involving both concatenation and finite-state transductions. Moreover, it has a completeness and termination guarantee for several important fragments (e.g. straight-line fragment). The main challenge addressed in the paper is the prohibitive worst-case complexity of the theory (double-exponential time), which is exponentially harder than the case without finite-state transductions. To this end, we propose a method that exploits succinct alternating finite-state automata as concise symbolic representations of string constraints. In contrast to previous approaches using nondeterministic automata, alternation offers not only exponential savings in space when representing Boolean combinations of transducers, but also a possibility of succinct representation of otherwise costly combinations of transducers and concatenation. Reasoning about the emptiness of the AFA language requires a state-space exploration in an exponential-sized graph, for which we use model checking algorithms (e.g. IC3). We have implemented our algorithm and demonstrated its efficacy on benchmarks that are derived from cross-site scripting analysis and other examples in the literature.

CCS Concepts: • **Theory of computation → Automated reasoning**; **Verification by model checking**; **Program verification**; **Program analysis**; *Logic and verification*; Complexity classes;

Authors' addresses: Lukáš Holík, Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Božetěchova 2, Brno, CZ-61266, Czech Republic, holik@fit.vutbr.cz; Petr Janků, Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Božetěchova 2, Brno, CZ-61266, Czech Republic, ijanku@fit.vutbr.cz; Anthony W. Lin, Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom, anthony.lin@cs.ox.ac.uk; Philipp Rümmer, Department of Information Technology, Uppsala University, Box 337, Uppsala, 75105, Sweden, philipp.ruemmer@it.uu.se; Tomáš Vojnar, Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Božetěchova 2, Brno, CZ-61266, Czech Republic, vojnar@fit.vutbr.cz.

2475-1421/2018/1-ART4
https://doi.org/10.1145/3158092

Additional Key Words and Phrases: String Solving, Alternating Finite Automata, Decision Procedure, IC3

ACM Reference Format:
Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. 2018. String Constraints with Concatenation and Transducers Solved Efficiently . *Proc. ACM Program. Lang.* 2, POPL, Article 4 (January 2018), 32 pages. https://doi.org/10.1145/3158092

1 INTRODUCTION

Strings are a fundamental data type in many programming languages. This statement is true now more than ever, especially owing to the rapidly growing popularity of scripting languages (e.g. JavaScript, Python, PHP, and Ruby) wherein programmers tend to make heavy use of string variables. String manipulations are often difficult to reason about automatically, and could easily lead to unexpected programming errors. In some applications, some of these errors could have serious security consequences, e.g., cross-site scripting (a.k.a. XSS), which are ranked among the top three classes of web application security vulnerabilities by OWASP [OWASP 2013].

Popular methods for analysing how strings are being manipulated by a program include *symbolic executions* [Bjørner et al. 2009; Cadar et al. 2008, 2011; Godefroid et al. 2005; Kausler and Sherman 2014; Loring et al. 2017; Redelinghuys et al. 2012; Saxena et al. 2010; Sen et al. 2013] which at their core use constraint solvers over the string domain (a.k.a. *string solvers*). String solvers have been the subject of numerous papers in the past decade, e.g., see [Abdulla et al. 2014; Balzarotti et al. 2008; Barrett et al. 2016; Bjørner et al. 2009; D'Antoni and Veanes 2013; Fu and Li 2010; Fu et al. 2013; Ganesh et al. 2013; Hooimeijer et al. 2011; Hooimeijer and Weimer 2012; Kiezun et al. 2012; Liang et al. 2014, 2016, 2015; Lin and Barceló 2016; Saxena et al. 2010; Trinh et al. 2014, 2016; Veanes et al. 2012; Wassermann et al. 2008; Yu et al. 2010, 2014, 2009, 2011; Zheng et al. 2013] among many others. As is common in constraint solving, we follow the standard approach of *Satisfiability Modulo Theories (SMT)* [De Moura and Bjørner 2011], which is an extension of the problem of satisfiability of Boolean formulae wherein each atomic proposition can be interpreted over some logical theories (typically, quantifier-free).

Unlike the case of constraints over integer/real arithmetic (where many decidability and undecidability results are known and powerful algorithms are already available, e.g., the simplex algorithm), string constraints are much less understood. This is because there are many different string operations that can be included in a theory of strings, e.g., concatenation, length comparisons, regular constraints (matching against a regular expression), and replace-all (i.e. replacing every occurrence of a string by another string). Even for the theory of strings with the concatenation operation alone, existing string solver cannot handle the theory (in its full generality) in a sound and complete manner, despite the existence of a theoretical decision procedure for the problem [Diekert 2002; Gutiérrez 1998; Jez 2016; Makanin 1977; Plandowski 2004, 2006]. This situation is exacerbated when we add extra operations like string-length comparisons, in which case even decidability is a long-standing open problem [Ganesh et al. 2013]. In addition, recent works in string solving have argued in favour of adding the replace-all operator or, more generally finite-state transducers, to string solvers [Lin and Barceló 2016; Trinh et al. 2016; Yu et al. 2010, 2014] in view of their importance for modelling relevant sanitisers (e.g. backslash-escape) and implicit browser transductions (e.g. an application of HTML-unescape by innerHTML), e.g., see [D'Antoni and Veanes 2013; Hooimeijer et al. 2011; Veanes et al. 2012] and Example 1.1 below. However, naively combining the replace-all operator and concatenation yields undecidability [Lin and Barceló 2016].

Example 1.1. The following JavaScript snippet—an adaptation of an example from [Kern 2014; Lin and Barceló 2016]—shows use of *both* concatenation and finite-state transducers:

```
var x = goog.string.htmlEscape(name);
var y = goog.string.escapeString(x);
nameElem.innerHTML = '<button onclick= "viewPerson(\'' + y + '\')">' + x + '</button>';
```

The code assigns an HTML markup for a button to the DOM element `nameElem`. Upon click, the button will invoke the function `viewPerson` on the input `name` whose value is an untrusted variable. The code attempts to first sanitise the value of `name`. This is done via The Closure Library [co 2015] string functions `htmlEscape` and `escapeString`. Inputting the value `Tom & Jerry` into `name` gives the desired HTML markup:

```
<button onclick="viewPerson('Tom &amp; Jerry')">Tom &amp; Jerry</button>
```

On the other hand, inputting value `');attackScript();//` to `name`, results in the markup:

```
<button onclick="viewPerson('&#39;);attackScript();//')">&#39;);attackScript();//')</button>
```

Before this string is inserted into the DOM via `innerHTML`, an implicit browser transduction will take place [Heiderich et al. 2013; Weinberger et al. 2011], i.e., HTML-unescaping the string inside the `onclick` attribute and then invoking the attacker's script `attackScript()` after `viewPerson`. This subtle DOM-based XSS bug is due to calling the right escape functions, but in wrong order. □

One theoretically sound approach proposed in [Lin and Barceló 2016] for overcoming the undecidability of string constraints with both concatenation and finite-state transducers is to impose a *straight-line restriction* on the shape of constraints. This straight-line fragment can be construed as the problem of *path feasibility* [Bjørner et al. 2009] in the following simple imperative language (with only assignment, skip, and assert) for defining non-branching and non-looping string-manipulating programs that are generated by symbolic execution:

$$S ::= y := a \mid \textbf{assert}(b) \mid \textbf{skip} \mid S_1; S_2, \qquad a ::= f(x_1, \ldots, x_n), \qquad b ::= g(x_1)$$

where $f : (\Sigma^*)^n \to \Sigma^*$ is either an application of concatenation $x_1 \circ \cdots \circ x_n$ or an application of a finite-state transduction $R(x_1)$, and g tests membership of x_1 in a regular language. Here, some variables are undefined "input variables". Path feasibility asks if there exist input strings that satisfy all assertions and applications of transductions in the program. It was shown in [Lin and Barceló 2016] that such a path feasibility problem (equivalently, satisfiability for the aforementioned straight-line fragment) is decidable. As noted in [Lin and Barceló 2016] such a fragment can express the program logic of many interesting examples of string-manipulating programs with/without XSS vulnerabilities. For instance, the above example can be modelled as a straight-line formula where the regular constraint comes from an attack pattern like the one below:

```
e1 = /<button onclick=
       "viewPerson\(' ( ' | [^']*[^'\\] ' ) \); [^']*[^'\\]' \)">.*<\/button>/
```

Unfortunately, the decidability proof given in [Lin and Barceló 2016] provides only a theoretical argument for decidability and complexity upper bounds (an exponential-time reduction to the *acyclic fragment* of intersection of rational relations[1] whose decidability proof in turn is a highly intricate polynomial-space procedure using Savitch's trick [Barceló et al. 2013]) and does not yield an implementable solution. Furthermore, despite its decidability, the string logic has a prohibitively high complexity (EXPSPACE-complete, i.e., exponentially higher than without transducers), which could severely limit its applicability.

[1]This fragment consists of constraints that are given as conjunctions of transducers $\bigwedge_{i=1}^{m} R_i(x_i, y_i)$, wherein the graph G of variables does not contain a cycle. The graph G contains vertices corresponding to variables x_i, y_i and that two variables x, y are linked by an edge if $x = x_i$ and $y = y_i$ for some $i \in \{1, \ldots, m\}$.

Contributions. Our paper makes the following contributions to overcome the above challenges:

(1) We propose a fast reduction of satisfiability of formulae in the straight-line fragment and in the acyclic fragment to the emptiness problem of *alternating finite-state automata (AFAs)*. The reduction is in the worst case exponential in the number of concatenation operations[2], but otherwise polynomial in the size of a formula. In combination with fast model checking algorithms (e.g. IC3 [Bradley 2012]) to decide AFA emptiness, this yields the first practical algorithm for handling string constraints with concatenation, finite-state transducers (hence, also replace-all), and regular constraints, and a decision procedure for formulae within the straight-line and acyclic fragments.
(2) We obtain a substantially simpler proof for the decidability and PSPACE-membership of the acyclic fragment of intersection of rational relations of [Barceló et al. 2013], which was crucially used in [Lin and Barceló 2016] as a blackbox in their decidability proof of the straight-line fragment.
(3) We define optimised translations from AFA emptiness to reachability over Boolean transition systems (i.e. which are succinctly represented by Boolean formulae). We implemented our algorithm for string constraints in a new string solver called Sloth, and provide an extensive experimental evaluation. Sloth is the first solver that can handle string constraints that arise from HTML5 applications with sanitisation and implicit browser transductions. Our experiments suggest that the translation to AFAs can circumvent the EXPSPACE worst-case complexity of the straight-line fragment in many practical cases.

An overview of the results. The main technical contribution of our paper is a new method for exploiting alternating automata (AFA) as a succinct symbolic representation for representing formulae in a complex string logic admitting concatenation and finite-state transductions. In particular, the satisfiability problem for the string logic is reduced to AFA language emptiness, for which we exploit fast model checking algorithms. Compared to previous methods [Abdulla et al. 2014; Lin and Barceló 2016] that are based on nondeterministic automata (NFA) and transducers, we show that AFA can incur *at most a linear blowup* for each string operation permitted in the logic (i.e. concatenation, transducers, and regular constraints). While the product NFA representing the intersection of the languages of two automata A_1 and A_2 would be of size $O(|A_1| \times |A_2|)$, the language can be represented using an AFA of size $|A_1| + |A_2|$ (e.g. see [Vardi 1995]). The difficult cases are how to deal with concatenation and replace-all, which are our contributions to the paper. More precisely, a constraint of the form $x := y.z \wedge x \in L$ (where L is the language accepted by an automaton A) was reduced in [Abdulla et al. 2014; Lin and Barceló 2016] to regular constraints on y and z by means of splitting A, which causes a cubic blow-up (since an "intermediate state" in A has to be guessed, and for each state a product of two automata has to be constructed). Similarly, taking the post-image $R(L)$ of L under a relation R represented by a finite-state transducer T gives us an automaton of size $O(|T| \times |A|)$. A naïve application of AFAs is not helpful for those cases, since also projections on AFAs are computationally hard.

The key idea to overcome these difficulties is to *avoid* applying projections altogether, and instead use the AFA to represent general k-ary *rational relations* (a.k.a. k-track finite-state transductions [Barceló et al. 2013; Berstel 1979; Sakarovitch 2009]). This is possible because we focus on formulae without negation, so that the (implicit) existential quantifications for applications of transducers can be placed outside the constraint. This means that our AFAs operate on alphabets that are exponential in size (for k-ary relations, the alphabet is $\{\epsilon, 0, 1\}^k$). To address this problem, we introduce a succinct flavour of AFA with symbolically represented transitions. Our definition is

[2]This is an unavoidable computational limit imposed by EXPSPACE-hardness of the problem [Lin and Barceló 2016].

similar to the concept of alternating symbolic automata in [D'Antoni et al. 2016] with one difference. While symbolic AFA take a transition $q \to_\psi \varphi$ from a state q to a set of states satisfying a formula φ if the input symbol satisfies a formula ψ, our succinct AFA can mix constraints on successor states with those on input symbols within a single transition formula (similarly to the symbolic transition representation of deterministic automata in MONA [Klarlund et al. 2002], where sets of transitions are represented as multi-terminal BDDs with states as terminal nodes). We show how automata splitting can be achieved with at most linear blow-up.

The succinctness of our AFA representation of string formulae is not for free since AFA language emptiness is a PSPACE-complete problem (in contrast to polynomial-time for NFA). However, modern model checking algorithms and heuristics can be harnessed to solve the emptiness problem. In particular, we use a linear-time reduction to reachability in Boolean transition systems similar to [Cox and Leasure 2017; Wang et al. 2016], which can be solved by state of the art model checking algorithms, such as IC3 [Bradley 2012], k-induction [Sheeran et al. 2000], or Craig interpolation-based methods [McMillan 2003], and tools like nuXmv [Cavada et al. 2014] or ABC [Brayton and Mishchenko 2010].

An interesting by-product of our approach is an efficient decision procedure for the acyclic fragment. The acyclic logic does not a priori allow concatenation, but is more liberal in the use of transducer constraints (which can encode complex relations like string-length comparisons, and the subsequence relation). In addition, such a logic is of interest in the investigation of complex path-queries for graph databases [Barceló et al. 2013; Barceló et al. 2012], which has been pursued independently of strings for verification. Our algorithm also yields an alternative and substantially simpler proof of PSPACE upper bound of the satisfiability problem of the logic.

We have implemented our AFA-based string solver as the tool Sloth, using the infrastructure provided by the SMT solver Princess [Rümmer 2008], and applying the nuXmv [Cavada et al. 2014] and ABC [Brayton and Mishchenko 2010] model checkers to analyse succinct AFAs. Sloth is a decision procedure for the discussed fragments of straight-line and acyclic string formulae, and is able to process SMT-LIB input with CVC4-style string operations, augmented with operations `str.replace`, `str.replaceall`[3], and arbitrary transducers defined using sets of mutually recursive functions. Sloth is therefore extremely flexible at supporting intricate string operations, including escape operations such as the ones discussed in Example 1.1. Experiments with string benchmarks drawn from the literature, including problems with replace, replace-all, and general transducers, show that Sloth can solve problems that are beyond the scope of existing solvers, while it is competitive with other solvers on problems with a simpler set of operations.

Organisation. We recall relevant notions from logic and automata theory in Section 2. In Section 3, we define a general string constraint language and mention several important decidable restrictions. In Section 4, we recall the notion of alternating finite-state automata and define a succinct variant that plays a crucial role in our decision procedure. In Section 5, we provide a new algorithm for solving the acyclic fragment of the intersection of rational relations using AFA. In Section 7, we provide our efficient reduction from the straight-line fragment to the acyclic fragment that exploits AFA constructions. To simplify the presentation of this reduction, we first introduce in Section 6 a syntactic sugar of the acyclic fragment called acyclic constraints with synchronisation parameters. In Section 8, we provide our reduction from the AFT emptiness to reachability in a Boolean transition system. Experimental results are presented in Section 9. Our tool Sloth can be obtained from https://github.com/uuverifiers/sloth/wiki. Finally, we conclude in Section 10. Missing proofs can be found in the full version.

[3] `str.replaceall` is the SMT-LIB syntax for the replace-all operation. On the other hand, `str.replace` represents the operation of replacing the *first* occurrence of the given pattern. In case there is no such occurrence, the string stays intact.

2 PRELIMINARIES

Logic. Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values, and A a set of Boolean variables. We write $\mathbb{F}_A$ to denote the set of *Boolean formulae* over A. In this context, we will sometimes treat subsets A' of A as the corresponding truth assignments $\{s \mapsto 1 \mid s \in A'\} \cup \{s \mapsto 0 \mid s \in A \setminus A'\}$ and write, for instance, $A' \models \varphi$ for $\varphi \in \mathbb{F}_A$ if the assignment satisfies φ. An *atom* is a Boolean variable; a *literal* is either a atom or its negation. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals, and in *negation normal form* (NNF) if negation only occurs in front of atoms. We denote the set of variables in a formula φ by $\mathrm{var}(\varphi)$. We use $\bar{x}$ to denote sequences $x_1, \ldots, x_n$ of length $|\bar{x}| = n$ of propositional variables, and we write $\varphi(\bar{x})$ to denote that $\bar{x}$ are the variables of φ. If we do not fix the order of the variables, we write $\varphi(X)$ for a formula with X being its set of variables. For a variable vector $\bar{x}$, we denote by $\{\bar{x}\}$ the set of variables in the vector.

We say that φ is *positive* (*negative*) on an atom $\alpha \in A$ if α appears under an even (odd) number of negations only. A formula that is positive (negative) on all its atoms is called positive (negative), respectively. The constant formulae `true` and `false` are both positive and negative. We use $\mathbb{F}_S^+$ and $\mathbb{F}_S^-$ to denote the sets of all positive and negative Boolean formulae over S, respectively.

Given a formula φ, we write $\widetilde{\varphi}$ to denote a formula obtained by replacing (1) every conjunction by a disjunction and vice versa and (2) every occurrence of `true` by `false` and vice versa. Note that $\widetilde{x} = x$, which means that $\widetilde{\varphi}$ is not the same as the negation of φ.

Strings and languages. Fix a finite alphabet Σ. Elements in Σ^* are interchangeably called words or strings, where the empty word is denoted by ϵ. The concatenation of strings u, v is denoted by $u \circ v$, occasionally just by uv to avoid notational clutter. We denote by $|w|$ the lenght of a word $w \in \Sigma^*$. For any word $w = a_1 \ldots a_n$, $n \geq 1$, and any index $1 \leq i \leq n$, we denote by $w[i]$ the letter a_i. A language is a subset of Σ^*. The concatenation of two languages L, L' is the language $L \circ L' = \{w \circ w' \mid w \in L \wedge w' \in L'\}$, and the iteration L^* of a language L is the smallest language closed under $\circ$ and containing L and ϵ.

Regular languages and rational relations. A regular language over a finite alphabet Σ is a subset of Σ^* that can be built by a finite number of applications of the operations of concatenation, iteration, and union from the languages $\{\epsilon\}$ and $\{a\}$, $a \in \Sigma$. An n-ary rational relation R over Σ is a subset of $(\Sigma^*)^n$ that can be obtained from a regular language L over the alphabet of n-tuples $(\Sigma \cup \{\epsilon\})^n$ as follows. Include $(w_1, \ldots, w_n)$ in R iff for some $(a_1^1, \ldots, a_n^1), \ldots, (a_1^k, \ldots, a_n^k) \in L$, $w_i = a_1 \circ \cdots \circ a_k$ for all $1 \leq i \leq n$. Here, $\circ$ is a concatenation over the alphabet Σ, and k denotes the length of the words w_i. In practice, regular languages and rational relations can be represented using various flavours of finite-state automata, which are discussed in detail in Section 4.

3 STRING CONSTRAINTS

We start by recalling a general string constraint language from [Lin and Barceló 2016] that supports concatenations, finite-state transducers, and regular expression matching. We will subsequently state decidable fragments of the language for which we design our decision procedure.

3.1 String Language

We assume a vocabulary of countably many *string variables* $x, y, z, \ldots$ ranging over Σ^*. A *string formula* over Σ is a Boolean combination φ of *word equations* $x = t$ whose right-hand side t might contain the concatenation operator, *regular constraints* $P(x)$, and *rational constraints* $\mathcal{R}(\bar{x})$:

$$\varphi \ ::= \ x = t \mid P(x) \mid \mathcal{R}(\bar{x}) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi, \qquad t \ ::= \ x \mid a \mid t \circ t \ .$$

In the grammar, x ranges over string variables, $\bar{x}$ over vectors of string variables, and $a \in \Sigma$ over letters. $R \subseteq (\Sigma^*)^n$ is assumed to be an n-ary rational relation on words of Σ^*, and $P \subseteq \Sigma^*$ is a regular

language. We will represent regular languages and rational relations by succinct automata and transducers denoted as $\mathcal{R}$ and $\mathcal{A}$, respectively. The automata and transducers will be formalized in Section 4. When the transducer $\mathcal{R}$ or automaton $\mathcal{A}$ representing a rational relation R or regular language P is known, we write $\mathcal{R}(\bar{x})$ or $\mathcal{A}(\bar{x})$ instead of $R(\bar{x})$ or $P(\bar{x})$ in the formulae, respectively.

A formula φ is interpreted over an *assignment* $\iota : \mathrm{var}(\varphi) \to \Sigma^*$ of its variables to strings over Σ^*. It *satisfies* φ, written $\iota \models \varphi$, iff the constraint φ becomes true under the substitution of each variable x by $\iota(x)$. We formalise the satisfaction relation for word equations, rational constraints, and regular constraints, assuming the standard meaning of Boolean connectives:

(1) ι satisfies the equation $x = t$ if $\iota(x) = \iota(t)$, extending ι to terms by setting $\iota(a) = a$ and $\iota(t_1 \circ t_2) = \iota(t_1) \circ \iota(t_2)$.
(2) ι satisfies the rational constraint $\mathcal{R}(x_1, \ldots, x_n)$ iff $(\iota(x_1), \ldots, \iota(x_n))$ belongs to $\mathcal{R}$.
(3) ι satisfies the regular constraint $P(x)$, for P a regular language, if and only if $\iota(x) \in P$.

A satisfying assignment for φ is also called a *solution* for φ. If φ has a solution, it is *satisfiable*.

The unrestricted string logic is undecidable, e.g., one can easily encode Post Correspondence Problem (PCP) as the problem of checking satisfiability of the constraint $\mathcal{R}(x, x)$, for some rational transducer $\mathcal{R}$ [Morvan 2000]. We therefore concentrate on practical decidable fragments.

3.2 Decidable Fragments

Our approach to deciding string formulae is based on two major insights. The first insight is that alternating automata can be used to efficiently decide positive Boolean combinations of rational constraints. This yields an algorithm for deciding (an extension of) the *acyclic fragment* of [Barceló et al. 2013]. The minimalistic definition of acyclic logic restricts rational constraints and does not allow word equations (in Section 5.1 a limited form of equations and arithmetic constraints over lengths will be shown to be encodable in the logic). Our definition of the acyclic logic AC below generalises that of [Barceló et al. 2013] by allowing k-ary rational constraints instead of *binary*.

Definition 3.1 (Acyclic formulae). Particularly, we say that a string formula φ is *acyclic* if it does not contain word equations, rational constraints $\mathcal{R}(x_1, \ldots, x_n)$ only appear positively and their variables $x_1, \ldots, x_n$ are pairwise distinct, and for every sub-formula $\psi \wedge \psi'$ at a positive position of φ (and also every $\psi \vee \psi'$ at a negative position) it is the case that $|\mathrm{free}(\psi) \cap \mathrm{free}(\psi')| \leq 1$, i.e., ψ and ψ' have *at most one* variable in common. We denote by AC the set of all acyclic formulae.

The second main insight we build on is that alternation allows a very efficient encoding of concatenation into rational constraints and automata (though only equisatisfiable, not equivalent). Efficient reasoning about concatenation combined with rational relations is the main selling point of our work from the practical perspective—this is what is most needed and was so far missing in applications like security analysis of web-applications. We follow the approach from [Lin and Barceló 2016] which defines so called straight-line conjunctions. Straight-line conjunctions essentially correspond to sequences of program assignments in the single static assignment form, possibly interleaved with assertions of regular properties. An equation $x = y_1 \circ \cdots \circ y_n$ is understood as an assignment to a program variable x. A rational constraint $\mathcal{R}(x, y)$ may be interpreted as an assignment to x as well, in which case we write it as $x = \mathcal{R}(y)$ (though despite the notation, $\mathcal{R}$ is not required to represent a function, it can still mean any rational relation).

Definition 3.2 (Straight-line conjunction). A conjunction of string constraints is then defined to be *straight-line* if it can be written as $\psi \wedge \bigwedge_{i=1}^{m} x_i = P_i$ where ψ is a conjunction of regular and negated regular constraints and each P_i is either of the form $y_1 \circ \cdots \circ y_n$, or $R(y)$ and, importantly, P_i cannot contain variables $x_i, \ldots, x_m$. We denote by SL the set of all straight-line conjunctions.

Example 3.3. The program snippet in Example 1.1 would be expressed as $x = \mathcal{R}_1(\text{name}) \wedge y = \mathcal{R}_2(x) \wedge z = w_1 \circ y \circ w_2 \circ x \circ w_3 \wedge u = \mathcal{R}_3(z)$. The transducers $\mathcal{R}_i$ correspond to the string operations at the respective lines: $\mathcal{R}_1$ is the htmlEscape, $\mathcal{R}_2$ is the escapeString, and $\mathcal{R}_3$ is the implicit transduction within innerHTML. Line 3 is translated into a conjunction of the concatenation and the third rational constraint encoding the implicit string operation at the assignment to innerHTML. In the concatenation, w_1, w_2, w_3 are words that correspond to the three constant strings concatenated with x and y on line 3. To test vulnerability, a regular constraint $\mathcal{A}(u)$ encoding the pattern e1 is added as a conjunct.

The fragment of straight-line conjunctions can be straightforwardly extended to disjunctive formulae. We say that a string formula is straight-line if every clause in its DNF is straight-line. A decision procedure for straight-line conjunctions immediately extends to straight-line formulae: instantiate the DPLL(T) framework [Nieuwenhuis et al. 2004] with a solver for straight-line conjunctions.

The straight-line and acyclic fragments are clearly syntactically incomparable: AC does not have equations, SL restricts more strictly combinations of rational relations and allows only binary ones. Regarding expressive power, SL can express properties which AC cannot: the straight-line constraint $x = yy$ cannot be expressed by any acyclic formula. On the other hand, whether or not AC formulae can be expressed in SL is not clear. Every AC formula can be expressed by a single n-ary acyclic rational constraint (c.f. Section 5), hence acyclic formulae and acyclic rational constraints are of the same power. It is not clear however whether straight-line formulae, which can use only binary rational constraints, can express arbitrary n-ary acyclic rational constraint.

4 SUCCINCT ALTERNATING AUTOMATA AND TRANSDUCERS

We introduce a succinct form of alternating automata and transducers that operate over *bit vectors*, i.e., functions $b : V \to \mathbb{B}$ where V is a finite, totally ordered set of bit variables. This is a variant of the recent automata model in [D'Antoni et al. 2016] that is tailored to our problem. Bit vectors can of course be described by strings over $\mathbb{B}$, conjunctions of literals over V, or sets of those elements $v \in V$ such that $b(v) = 1$. In what follows, we will use all of these representations interchangeably. Referring to the last mentioned possibility, we denote the set of all bit vectors over V by $\mathcal{P}(V)$.

An obvious advantage of this approach is that encoding symbols of large alphabets, such as UTF, by bit vectors allows one to succinctly represent sets of such symbols using Boolean formulae. In particular, symbols of an alphabet of size 2^k can be encoded by bit vectors of size k (or, alternatively, as Boolean formulae over k Boolean variables). We use this fact when encoding transitions of our alternating automata.

Example 4.1. To illustrate the encoding, assume the alphabet $\Sigma = \{a, b, c, d\}$ consisting of symbols a, b, c, and d. We can deal with this alphabet by using the set $V = \{v_0, v_1\}$ and representing, e.g., a as $\neg v_1 \wedge \neg v_0$, b as $\neg v_1 \wedge v_0$, c as $v_1 \wedge \neg v_0$, and d as $v_1 \wedge v_0$. This is, a, b, c, and d are encoded as the bit vectors 00, 01, 10, and 11 (for the ordering $v_0 < v_1$), or the sets $\emptyset$, $\{v_0\}$, $\{v_1\}$, $\{v_0, v_1\}$, respectively. The set of symbols $\{c, d\}$ can then be encoded simply by the formula v_1. □

4.1 Succinct Alternating Finite Automata

A *succinct alternating finite automaton (AFA)* over Boolean variables V is a tuple $\mathcal{A} = (V, Q, \Delta, I, F)$ where Q is a finite set of *states*, the *transition function* $\Delta : Q \to \mathbb{F}_{V \cup Q}$ assigns to every state a Boolean formula over Boolean variables and states that is positive on states, $I \in \mathbb{F}^+_Q$ is a positive *initial formula*, and $F \in \mathbb{F}^-_Q$ is a negative *final formula*. Let $w = b_1 \ldots b_m$, $m \geq 0$, be a word where each b_i, $1 \leq i \leq m$, is a bit vector encoding the i-th letter of w. A *run* of the AFA $\mathcal{A}$ over w is a sequence $\rho = \rho_0 b_1 \rho_1 \ldots b_m \rho_m$ where $b_i \in \mathcal{P}(V)$ for every $1 \leq i \leq m$, $\rho_i \subseteq Q$ for every $0 \leq i \leq m$, and

$b_i \cup \rho_i \models \bigwedge_{q \in \rho_{i-1}} \Delta(q)$ for every $1 \leq i \leq m$. The run is *accepting* if $\rho_0 \models I$ and $\rho_m \models F$, in which case the word is accepted. The *language* of $\mathcal{A}$ is the set $L(\mathcal{A})$ of accepted words.

Notice that instead of the more usual definition of Δ, which would assign a positive Boolean formula over Q to every pair from $Q \times \mathcal{P}(V)$ or to a pair $Q \times \mathbb{F}_V$ as in [D'Antoni et al. 2016], we let Δ assign to states formulae that talk about both target states and Boolean input variables. This is closer to the encoding of the transition function as used in MONA [Klarlund et al. 2002]. It allows for additional succinctness and also for a more natural translation of the language emptiness problem into a model checking problem (cf. Section 8).[4] Moreover, compared with the usual AFA definition, we do not have just a single initial state and a single set of accepting states, but we use initial and final formulae. As will become clear in Section 5, this approach allows us to easily translate the considered formulae into AFAs in an inductive way.

Note that standard *nondeterministic finite automata* (NFAs), working over bit vectors, can be obtained as a special case of our AFAs as follows. An AFA $\mathcal{A} = (V, Q, \Delta, I, F)$ is an NFA iff (1) I is of the form $\bigvee_{q \in Q'} q$ for some $Q' \subseteq Q$, (2) F is of the form $\bigwedge_{q \in Q''} \neg q$ for some $Q'' \subseteq Q$, and (3) for every $q \in Q$, $\Delta(q)$ is of the form $\bigvee_{1 \leq i \leq m} \varphi_i(V) \wedge q_i$ where $m \geq 0$ and, for all $1 \leq i \leq m$, $\varphi_i(V)$ is a formula over the input bit variables and $q_i \in Q$.

Example 4.2. To illustrate our notion of AFAs, we give an example of an AFA $\mathcal{A}$ over the alphabet $\Sigma = \{a, b, c, d\}$ from Example 4.1 that accepts the language $\{w \in \Sigma^* \mid |w| \bmod 35 = 0 \wedge \forall i \exists j : (1 \leq i \leq |w| \wedge w[i] \in \{a, b\}) \rightarrow (i < j \leq |w| \wedge w[j] \in \{c, d\})\}$, i.e., the length of the words is a multiple of 35, and every letter a or b is eventually followed by a letter c or d. In particular, we let $\mathcal{A} = (\{v_0, v_1\}, \{q_0, \ldots, q_4, p_0, \ldots, p_6, r_1, r_2\}, \Delta, I, F)$ where $I = q_0 \wedge p_0$, $F = \neg q_1 \wedge \ldots \wedge \neg q_4 \wedge \neg p_1 \wedge \ldots \wedge \neg p_6 \wedge \neg r_1$ (i.e., the accepting states are q_0, p_0, and r_2), and Δ is defined as follows:

- $\forall 0 \leq i < 5 : \Delta(q_i) = (\neg v_1 \wedge q_{(i+1) \bmod 5} \wedge r_1) \vee (v_1 \wedge q_{(i+1) \bmod 5})$,
- $\forall 0 \leq i < 7 : \Delta(p_i) = p_{(i+1) \bmod 7}$,
- $\Delta(r_1) = (v_1 \wedge r_2) \vee (\neg v_1 \wedge r_1)$ and $\Delta(r_2) = r_2$.

Intuitively, the q states check divisibility by 5. Moreover, whenever, they encounter an a or b symbol (encoded succinctly as checking $\neg v_1$ in the AFA), they spawn a run through the r states, which checks that eventually a c or d symbol appears. The p states then check divisibility by 7. The desired language is accepted due to the requirement that all these runs must be synchronized. Note that encoding the language using an NFA would require quadratically more states since an explicit product of all the branches would have to be done. □

The additional succinctness of AFA does not influence the computational complexity of the emptiness check compared to the standard variant of alternating automata.

Lemma 4.3. *The problem of language emptiness of AFA is PSPACE-complete.*

The lemma is witnessed by a linear-space transformation of the problem of emptiness of an AFA language to the PSPACE-complete problem of reachability in a Boolean transition system. This transformation is shown in Section 8.

4.2 Boolean Operations on AFAs

From the standard Boolean operations over AFAs, we will mainly need conjunction and disjunction in this paper. These operations can be implemented in linear space and time in a way analogous to [D'Antoni et al. 2016], slightly adapted for our notion of initial/final formulae, as follows. Given

[4] [D'Antoni et al. 2016] also mentions an implementation of symbolic AFAs that uses MONA-like BDDs and is technically close to our AFAs.

two AFAs $\mathcal{A} = (V, Q, \Delta, I, F)$ and $\mathcal{A}' = (V, Q', \Delta', I', F')$ with $Q \cap Q' = \emptyset$, the automaton accepting the union of their languages can be constructed as $\mathcal{A} \cup \mathcal{A}' = (V, Q \cup Q', \Delta \cup \Delta', I \vee I', F \wedge F')$, and the automaton accepting the intersection of their languages can be constructed as $\mathcal{A} \cap \mathcal{A}' = (V, Q \cup Q', \Delta \cup \Delta', I \wedge I', F \wedge F')$. Seeing correctness of the construction of $\mathcal{A} \cap \mathcal{A}'$ is immediate. Indeed, the initial condition enforces that the two AFAs run in parallel, disjointness of their state-spaces prevents them from influencing one another, and the final condition defines their parallel runs as accepting iff both of the runs accept. To see correctness of the construction of $\mathcal{A} \cup \mathcal{A}'$, it is enough to consider that one of the automata can be started with the empty set of states (corresponding to the formula $\bigwedge_{q \in Q} \neg q$ for $\mathcal{A}$ and likewise for $\mathcal{A}'$). This is possible since only one of the initial formulae I and I' needs to be satisfied. The automaton that was started with the empty set of states will stay with the empty set of states throughout the entire run and thus trivially satisfy the (negative) final formula.

Example 4.4. Note that the AFA in Example 4.2 can be viewed as obtained by conjunction of two AFAs: one consisting of the q and r states and the second of the p states. □

To complement an AFA $\mathcal{A} = (V, Q, \Delta, I, F)$, we first transform the automaton into a form corresponding to the symbolic AFA of [D'Antoni et al. 2016] and then use their complementation procedure. More precisely, the transformation to the symbolic AFA form requires two steps:

- The first step simplifies the final condition. The final formula F is converted into DNF, yielding a formula $F_1 \vee \ldots \vee F_k$, $k \geq 1$, where each F_i, $1 \leq i \leq k$, is a conjunction of negative literals over Q. The AFA $\mathcal{A}$ is then transformed into a union of AFAs $\mathcal{A}_i = (V, Q, \Delta, I, F_i)$, $1 \leq i \leq k$, where each $\mathcal{A}_i$ is a copy of $\mathcal{A}$ except that it uses one of the disjuncts F_i of the DNF form of the original final formula F as its final formula. Each resulting AFAs hence have a purely conjunctive final condition that corresponds a set of final states of [D'Antoni et al. 2016] (a set of final states $F \subseteq Q$ would correspond to the final formula $\bigwedge_{q \in Q \setminus F} \neg q$).
- The second step simplifies the structure of the transitions. For every $q \in Q$, the transition formula $\Delta(q)$ is transformed into a disjunction of formulae of the form $(\varphi_1(V) \wedge \psi_1(Q)) \vee \ldots \vee (\varphi_m(V) \wedge \psi_m(Q))$ where the $\varphi_i(V)$ formulae, called *input formulae* below, speak about input bit variables only, while the $\psi_i(Q)$ formulae, called *target formulae* below, speak exclusively about the target states, for $1 \leq i \leq m$. For this transformation, a slight modification of transforming a formula into DNF can be used.

The complementation procedure of [D'Antoni et al. 2016] then proceeds in two steps: the *normalisation* and the complementation itself. We sketch them below:

- For every $q \in Q$, normalisation transforms the transition formula $\Delta(q) = (\varphi_1(V) \wedge \psi_1(Q)) \vee \ldots \vee (\varphi_m(V) \wedge \psi_m(Q))$ so that every two distinct input formulae $\varphi(V)$ and $\varphi'(V)$ of the resulting formula describe disjoint sets of bit vectors, i.e., $\neg(\varphi(V) \wedge \varphi'(V))$ holds. To achieve this (without trying to optimize the algorithm as in [D'Antoni et al. 2016]), one can consider generating all Boolean combinations of the original $\varphi(V)$ formulae, conjoining each of them with the disjunction of those state formulae whose input formulae are taken positively in the given case. More precisely, one can take $\bigvee_{I \subseteq \{1,\ldots,m\}} (\bigwedge_{i \in I} \varphi_i)) \wedge (\bigwedge_{i \in \{1,\ldots,m\} \setminus I} \neg \varphi_i)) \wedge \bigvee_{i \in I} \psi_i$.
- Finally, to complement the AFAs normalized in the above way, one proceeds as follows: (1) The initial formula I is replaced by $\widetilde{I}$. (2) For every $q \in Q$ and every disjunct $\varphi(V) \wedge \psi(Q)$ of the transition formula $\Delta(q)$, the target formula $\psi(Q)$ is replaced by $\widetilde{\psi}(Q)$. (3) The final formula of the form $\bigwedge_{q \in Q'} \neg q$, $Q' \subseteq Q$, is transformed to the formula $\bigwedge_{q \in Q \setminus Q'} \neg q$, and `false` is swapped for `true` and vice versa.

Clearly, the complementation contains three sources of exponential blow-up: (1) the simplification of the final condition, (2) the simplification of transitions and (3) the normalization of transitions.

Note, however, that, in this paper, we will apply complementation exclusively on AFAs obtained by Boolean operations from NFAs derived from regular expressions. Such AFAs already have the simple final conditions, and so the first source of exponential blow-up does not apply. The second and the third source of exponential complexity can manifest themselves but note that it does not show up in the number of states. Finally, note that if we used AFAs with explicit alphabets, the second and the third problem would disappear (but then the AFAs would usually be bigger anyway).

4.3 Succinct Alternating Finite Transducers

In our alternating finite transducers, we will need to use *epsilon symbols* representing the empty word. Moreover, as we will explain later, in order to avoid some undesirable synchronization when composing the transducers, we will need more such symbols—differing just syntactically. Technically, we will encode the epsilon symbols using a set of epsilon bit variables E, containing one new bit variable for each epsilon symbol. We will draw the epsilon bit variables from a countably infinite set $\mathcal{E}$. We will also assume that when one of these bits is set, other bits are not important.

Let W be a finite, totally ordered set of bit variables, which we can split to the set of input bit variables $V(W) = W \setminus \mathcal{E}$ and the set of epsilon bit variables $E(W) = W \cap \mathcal{E}$. Given a word $w = b_1 \dots b_m \in \mathcal{P}(W)^*$, $m \geq 0$, we denote by $\rangle w \langle$ the word that arises from w by erasing all those b_i, $1 \leq i \leq m$, in which some epsilon bit variable is set, i.e., $b_i \cap \mathcal{E} \neq \emptyset$. Further, let $k \geq 1$, and let $W\langle k \rangle = W \times [k]$, assuming it to be ordered in the lexicographic way. The indexing of the bit variables will be used to express the track on which they are read. Finally, given a word $w = b_1 \dots b_m \in \mathcal{P}(W\langle k \rangle)^*$, $m \geq 0$, we denote by $w\downarrow_i$, $1 \leq i \leq k$, the word $b'_1 \dots b'_m \in \mathcal{P}(W)^*$ that arises from w by keeping the contents of the i-th track (without the index i) only, i.e., $b'_j \times \{i\} = b_j \cap (W \times \{i\})$ for $1 \leq j \leq m$.

A k-track *succinct alternating finite transducer* (AFT) over W is syntactically an alternating automaton $\mathcal{R} = (W\langle k \rangle, Q, \Delta, I, F)$, $k \geq 1$. Let $V = V(W)$. The relation $Rel(\mathcal{R}) \subseteq (\mathcal{P}(V)^*)^k$ *recognised* by $\mathcal{R}$ contains a k-tuple of words $(x_1, \dots, x_k)$ over $\mathcal{P}(V)$ iff there is a word $w \in L(\mathcal{R})$ such that $x_i = \rangle w\downarrow_i \langle$ for each $1 \leq i \leq k$.

Below, we will sometimes say that the word w encodes the k-tuple of words $(x_1, \dots, x_k)$. Moreover, for simplicity, instead of saying that $\mathcal{R}$ has a run over w that encodes $(x_1, \dots, x_k)$, we will sometimes directly say that $\mathcal{R}$ has a run over $(x_1, \dots, x_k)$ or that $\mathcal{R}$ accepts $(x_1, \dots, x_k)$.

Finally, note that classical *nondeterministic finite transducers* (NFTs) are a special case of our AFTs that can be defined by a similar restriction as the one used when restricting AFAs to NFAs. In particular, the first track (with letters indexed with 1) can be seen as the input track, and the second track (with letters indexed with 2) can be seen as the output track. AFTs as well as NFTs recognize the class of *rational relations* [Barceló et al. 2013; Berstel 1979; Sakarovitch 2009].

Example 4.5. We now give a simple example of an AFT that implements escaping of every apostrophe by a backlash in the UTF-8 encoding. Intuitively, the AFT will transform an input string `x'xx` to the string `x\'xx`, i.e., the relation it represents will contain the couple (`x'xx`,`x\'xx`). All the symbols should, however, be encoded in UTF-8. In this encoding, the apostrophe has the binary code 00100111, and the backlash has the code 00101010. We will work with the set of bit variables $V_8 = \{v_0, \dots, v_7\}$ and a single epsilon bit variable e. We will superscript the bit variables by the track on which they are read (hence, e.g., v_1^2 is the same as $(v_1, 2)$, i.e., v_1 is read on the second track). Let $\mathsf{ap}^i = v_0^i \wedge v_1^i \wedge v_2^i \wedge \neg v_3^i \wedge \neg v_4^i \wedge v_5^i \wedge \neg v_6^i \wedge \neg v_7^i \wedge \neg e^i$ represent an apostrophe read on the i-th track. Next, let $\mathsf{bc}^i = \neg v_0^i \wedge v_1^i \wedge \neg v_2^i \wedge v_3^i \wedge \neg v_4^i \wedge v_5^i \wedge \neg v_6^i \wedge \neg v_7^i \wedge \neg e^i$ represent a backlash read on the i-th track. Finally, let $\mathsf{eq}^{i,j} = e^i \leftrightarrow e^j \wedge \bigwedge_{0 \leq k < 8} v_k^i \leftrightarrow v_k^j$ denote that the same symbol is read on the i-th and j-th track. The AFT that implements the described escaping can

be constructed as follows: $\mathcal{R} = ((V_8 \cup \{e\})\langle 2\rangle, \{q_0, q_1\}, \Delta, q_0, \neg q_1)$ where the transition formulae are defined by $\Delta(q_0) = (\neg \mathsf{ap}^1 \wedge \mathsf{eq}^{1,2} \wedge q_0) \vee (\mathsf{ap}^1 \wedge \mathsf{bc}^2 \wedge q_1)$ and $\Delta(q_1) = e^1 \wedge \mathsf{ap}^2 \wedge q_0$. □

5 DECIDING ACYCLIC FORMULAE

Our decision procedure for AC formulae is based on translating them into AFTs. For simplicity, we assume that the formula is negation free (after transforming to NNF, negation at regular constraints can be eliminated by AFA complementation). Notice that with no negations, the restriction AC puts on disjunctions never applies. We also assume that the formula contains rational constraints only (regular constraint can be understood as unary rational constraints).

Our algorithm then transforms a formula $\varphi(\bar{x})$ into a rational constraint $\mathcal{R}_\varphi(\bar{x})$ inductively on the structure of φ. As the base case, we get rational constraints $\mathcal{R}(\bar{x})$, which are already represented as AFTs, and regular constraints $\mathcal{A}(x)$, already represented by AFAs. Boolean operations over regular constraints can be treated using the corresponding Boolean operations over AFAs described in Section 4.2. The resulting AFAs can then be viewed as rational constraints with one variable (and hence as a single-track AFT).

Once constraints $\mathcal{R}_\varphi(\bar{x})$ and $\mathcal{R}_\psi(\bar{y})$ are available, the induction step translates formulae $\mathcal{R}_\varphi(\bar{x}) \wedge \mathcal{R}_\psi(\bar{y})$ and $\mathcal{R}_\varphi(\bar{x}) \vee \mathcal{R}_\psi(\bar{y})$ to constraints $\mathcal{R}_{\varphi\wedge\psi}(\bar{z})$ and $\mathcal{R}_{\varphi\vee\psi}(\bar{z})$, respectively. To be able to describe this step in detail, let $\mathcal{R}_\varphi = ((V \cup E_\varphi)\langle|\bar{x}|\rangle, Q_\varphi, \Delta_\varphi, I_\varphi, F_\varphi)$ and $\mathcal{R}_\psi = ((V \cup E_\psi)\langle|\bar{y}|\rangle, Q_\psi, \Delta_\psi, I_\psi, F_\psi)$ such that w.l.o.g. $Q_\varphi \cap Q_\psi = \emptyset$ and $E_\varphi \cap E_\psi = \emptyset$.

Translation of conjunctions to AFTs. The construction of $\mathcal{R}_{\varphi\wedge\psi}$ has three steps:

(1) **Alignment of tracks** that ensures that distinct variables are assigned different tracks and that the transducers agree on the track used for the shared variable.
(2) **Saturation by ϵ-self loops** allowing the AFTs to synchronize whenever one of them makes an ϵ move on the shared track.
(3) **Conjunction** on the resulting AFTs viewing them as AFAs.

Alignment of tracks. Given constraints $\mathcal{R}_\varphi(\bar{x})$ and $\mathcal{R}_\psi(\bar{y})$, the goal of the alignment of tracks is to assign distinct tracks to distinct variables of $\bar{x}$ and $\bar{y}$, and to assign the same track in both of the transducers to the shared variable—if there is one (recall that, by acyclicity, $\bar{x}$ and $\bar{y}$ do not contain repeating variables and share at most one common variable). This is implemented by choosing a vector $\bar{z}$ that consists of exactly one occurrence of every variable from $\bar{x}$ and $\bar{y}$, i.e., $\{\bar{z}\} = \{\bar{x}\} \cup \{\bar{y}\}$, and by subsequently re-indexing the bit vector variables in the transition relations. Particularly, in Δ_φ, every indexed bit vector variable v^i (including epsilon bit variables) is replaced by v^j with j being the position of x_i in $\bar{z}$, and analogously in Δ_ψ, every indexed bit variable v^i is replaced by v^j with j being the position of y_i in $\bar{z}$. Both AFTs are then considered to have $|\bar{z}|$ tracks.

Saturation by ϵ-self loops. This step is needed if $\bar{x}$ and $\bar{y}$ share a variable, i.e., $\{\bar{x}\} \cap \{\bar{y}\} \neq \emptyset$. The two input transducers then have to synchronise on reading its symbols. However, it may happen that, at some point, one of them will want to read from the non-shared tracks exclusively, performing an ϵ transition on the shared track. Since reading of the non-shared tracks can be ignored by the other transducer, it should be allowed to perform an ϵ move on all of its tracks. However, that needs not be allowed by its transition function. To compensate for this, we will saturate the transition function by ϵ-self loops performed on all tracks. Unfortunately, there is one additional problem with this step: If the added ϵ transitions were based on the same epsilon bit variables as those already used in the given AFT, they could enable some additional synchronization *within* the given AFT, thus allowing it to accept some more tuples of words. We give an example of this problem below (Example 5.2). To resolve the problem, we assume that the two AFTs being conjuncted use different epsilon bit variables (more of such variables can be used due the AFTs can be a result of

several previous conjunctions). Formally, for any choice $\sigma, \sigma' \in \{\varphi, \psi\}$ such that $\sigma \neq \sigma'$, and for every state $q \in Q_\sigma$, the transition formula $\Delta_\sigma(q)$ is replaced by $\Delta_\sigma(q) \vee (q \wedge \bigvee_{e \in E_{\sigma'}} \bigwedge_{i \in [|\bar{z}|]} e^i)$.

Conjunction of AFTs viewed as AFAs. In the last step, the input AFTs with aligned tracks and saturated by ϵ-self loops are conjoined using the automata intersection construction from Section 4.2.

Lemma 5.1. *Let $\mathcal{R}'_\varphi$ and $\mathcal{R}'_\psi$ be the AFTs obtained from the input AFTs $\mathcal{R}_\varphi$ and $\mathcal{R}_\psi$ by track alignment and ϵ-self-loop saturation, and let $\mathcal{R}_{\varphi\wedge\psi} = \mathcal{R}'_\varphi \cap \mathcal{R}'_\psi$. Then, $\mathcal{R}_{\varphi\wedge\psi}(\bar{z})$ is equivalent to $\mathcal{R}_\varphi(\bar{x}) \wedge \mathcal{R}_\psi(\bar{y})$.*

To see that the lemma holds, note that both $\mathcal{R}'_\varphi$ and $\mathcal{R}'_\psi$ have the same number of tracks—namely, $|\bar{z}|$. This number can be bigger than the original number of tracks ($|\bar{x}|$ or $|\bar{y}|$, resp.), but the AFTs still represent the same relations over the original tracks (the added tracks are unconstrained). The ϵ-self loop saturation does not alter the represented relations either as the added transitions represent empty words across all tracks only, and, moreover, they cannot synchronize with the original transitions, unblocking some originally blocked runs. Finally, due to the saturation, the two AFTs cannot block each other by an epsilon move on the shared track available in one of them only.[5]

Example 5.2. We now provide an example illustrating the conjunction of AFTs, including the need to saturate the AFTs by ϵ-self loops with different ϵ symbols. We will assume working with the input alphabet $\Sigma = \{a, b\}$ encoded using a single input bit variable v_0: let a correspond to $\neg v_0$ and b to v_0. Moreover, we will use two epsilon bit variables, namely, e_1 and e_2. We consider the following two simple AFTs, each with two tracks:

- $\mathcal{R}_1 = (\{v_0, e_1\}\langle 2\rangle, \{q_0, q_1, q_2\}, \Delta_1, q_0, \neg q_0 \wedge \neg q_2)$ with $\Delta_1(q_0) = (a^1 \wedge b^2 \wedge q_1) \vee (a^1 \wedge a^2 \wedge q_1 \wedge q_2)$, $\Delta_1(q_1) = \texttt{false}$, and $\Delta_1(q_2) = e_1^1 \wedge q_1$. Note that $Rel(\mathcal{R}_1) = \{(a, b)\}$ since the run that starts with $a^1 \wedge a^2$ gets stuck in one of its branches, namely the one that goes to q_2. This is because we require branches of a single run of an AFT to synchronize even on epsilon bit variables, and the transition from q_2 cannot synchronize with any move from q_1.
- $\mathcal{R}_2 = (\{v_0, e_2\}\langle 2\rangle, \{p_0, p_1, p_2\}, \Delta_2, p_0, \neg p_0 \wedge \neg p_1)$ such that $\Delta_2(p_0) = (a^1 \wedge b^2 \wedge p_1)$, $\Delta_2(p_1) = e_2^1 \wedge b^2 \wedge p_2$, and $\Delta_2(p_2) = \texttt{false}$. Clearly, $Rel(\mathcal{R}_2) = \{(a, bb)\}$.

Let Q_i, I_i, F_i denote the set of states, initial constraint, and final constraint of $\mathcal{R}_i$, $i \in \{1, 2\}$, respectively. Assume that we want to construct an AFT for the constraint $\mathcal{R}_1(x, y) \wedge \mathcal{R}_2(x, z)$. This constraint represents the ternary relation $\{(a, b, bb)\}$. It can be seen that if we apply the above described construction for intersection of AFTs to $\mathcal{R}'_1$ and $\mathcal{R}'_2$, where $\mathcal{R}'_1 = \mathcal{R}_1$ and $\mathcal{R}'_2$ is the same as $\mathcal{R}_2$ up to all symbols from track to 2 are moved to track 3, we will get an AFT $\mathcal{R} = (\{v_0, e_1, e_2\}\langle 3\rangle, Q_1 \cup Q_2, \Delta, I_1 \wedge I_2, F_1 \wedge F_2)$ representing exactly this relation. We will not list here the entire Δ but let us note the below:

- Δ will contain the following transition obtained by ϵ-self-loop saturation of $\mathcal{R}_1$: $\Delta(q_1) = (e_2^1 \wedge e_2^2 \wedge q_1)$. This will allow $\mathcal{R}$ to synchronize its run through q_1 with its run from p_1 to p_2. Without the saturation, this would not be possible, and $Rel(\mathcal{R})$ would be empty.
- On the other hand, if a single epsilon bit variable e was used in both AFTs as well as in their saturation, the saturated Δ_1 would include the transition $\Delta_1(q_1) = (e^1 \wedge e^2 \wedge q_1)$. This transition could synchronize with the transition $\Delta_1(q_2) = e^1 \wedge q_1$, and the relation represented by the saturated $\mathcal{R}_1$ would grow to $Rel(\mathcal{R}_1) = \{(a, b), (a, a)\}$. The result of the intersection would then (wrongly) represent the relation $\{(a, b, bb), (a, a, bb)\}$. □

[5]Note that the same approach cannot be used for AFTs sharing more than one track. Indeed, by intersecting two general rational relations, one needs not obtain a rational relation.

Translation of disjunctions to AFTs. The construction of an AFT for a disjunction of formulae is slightly simpler. The alignment of variables is immediately followed by an application of the AFA disjunction construction. That is, the AFT $\mathcal{R}_{\varphi\vee\psi}$ is constructed simply as $\mathcal{R}'_\varphi \cup \mathcal{R}'_\psi$ from the constraints $\mathcal{R}'_\varphi(\bar{z})$ and $\mathcal{R}'_\psi(\bar{z})$ produced by the alignment of the vectors of variables $\bar{x}$ and $\bar{y}$ in $\mathcal{R}_\varphi(\bar{x})$ and $\mathcal{R}_\psi(\bar{y})$. The construction of $\mathcal{R}'_\varphi$ and $\mathcal{R}'_\psi$ does not require the saturation by ϵ-self loops because the two transducers do not need to synchronise on reading shared variables. The vectors $\bar{x}$ and $\bar{y}$ are allowed to share any number of variables.

Theorem 5.3. *Every acyclic formula $\varphi(\bar{x})$ can be transformed into an equisatisfiable rational constraint $\mathcal{R}(\bar{x})$ represented by an AFT $\mathcal{R}$. The transformation can be done in polynomial time unless φ contains a negated regular constraint represented by a non-normalized succinct NFA.*

Corollary 5.4. *Checking satisfiability of acyclic formulae is in PSPACE unless the formulae contain a negated regular constraint represented by a non-normalized succinct NFA.*

PSPACE membership of satisfiability of acyclic formulae with binary rational constraints (without negations of regular constraints and without considering succinct alphabet encoding) is proven already in [Barceló et al. 2013]. Apart from extending the result to k-ary rational constraints, we obtain a simpler proof as a corollary of Theorem 5.3, avoiding a need to use the highly intricate polynomial-space procedure based on the Savitch´s trick used in [Barceló et al. 2013]. Not considering the problem of negating regular constraints, our PSPACE algorithm would first construct a linear-size AFT for the input φ. We can then use the fact that the standard PSPACE algorithm for checking emptiness of AFAs/AFTs easily generalises to succinct AFAs/AFTs. This is proved by our linear-space reduction of emptiness of the language of succinct AFAs to reachability in Boolean transition systems, presented in Section 8. Reachability in Boolean transition systems is known to be PSPACE-complete.

5.1 Decidable Extensions of AC

The relatively liberal condition that AC puts on rational constraints allow us to easily extend AC with other features, without having to change the decision procedure. Namely, we can add Presburger constraints about word length, as well as word equations, as long as overall acyclicity of a formula is preserved. Length constraints can be added in the general form $\varphi_{\text{Pres}}(|x_1|, \ldots, |x_k|)$, where φ_{Pres} is a Presburger formula.

Definition 5.5 (Extended acyclic formulae). A string formula φ augmented with length constraints $\varphi_{\text{Pres}}(|x_1|, \ldots, |x_k|)$ is *extended acyclic* if every word equation or rational constraint contains each variable at most once, rational constraints $\mathcal{R}(x_1, \ldots, x_n)$ only appear at positive positions, and for every sub-formula $\psi \wedge \psi'$ at a positive position of φ (and also every $\psi \vee \psi'$ at a negative position) it is the case that $|\text{free}(\psi) \cap \text{free}(\psi')| \leq 1$, i.e., ψ and ψ' have *at most one* variable in common.

Any extended AC formula φ can be turned into a standard AC formula by translating word equations and length constraints to rational constraints. Notice that, although quite powerful, extended AC still cannot express SL formulae such as $x = yy$, and does not cover practical properties such as, e.g., those in Example 3.3 (where two conjuncts contain both x and y).

Word equations to rational constraints. For simplicity, assume that equations do not contain letters $a \in \Sigma$. This can be achieved by replacing every occurrence of a constraint b by a fresh variable constrained by the regular language $\{b\}$. An equation $x = x_1 \circ \cdots \circ x_n$ without multiple occurrences of any variables is translated to a rational constraint $\mathcal{R}(x, x_1, \ldots, x_n)$ with $\mathcal{R} = (W\langle n+1\rangle, Q =$

$\{q_0, \ldots, q_n\}, \Delta, I = q_0, F = q_n)$. The transitions for $i \in [n]$ are

$$\Delta(q_{i-1}) = (q_{i-1} \vee q_i) \wedge \bigwedge_{j \in [n] \setminus \{i\}} e^j \wedge \bigwedge_{v \in W\langle n+1 \rangle} (v^i \leftrightarrow v^0).$$

and $\Delta(q_n) = \texttt{false}$. That is, the symbol on the first track is copied to the ith track while all the other tracks read ϵ. Negated word equations can be translated to AFTs in a similar way.

Length constraints to rational constraints. The translation of length constraints to rational constraints is similarly straightforward. Suppose an extended AC formula contains a length constraint $\varphi_{\text{Pres}}(|x_1|, \ldots, |x_k|)$, where φ_{Pres} is a Presburger formula over k variables $y_1, \ldots, y_k$ ranging over natural numbers. It is a classical result that the solution space of φ_{Pres} forms a semi-linear set [Ginsburg and Spanier 1966], i.e., can be represented as a finite union of linear sets $L_j = \{\bar{y}_0 + \sum_{i=1}^{m} \lambda_i \bar{y}_i \mid \lambda_1, \ldots, \lambda_m \in \mathbb{N}\} \subseteq \mathbb{N}^k$ with $\bar{y}_0, \ldots \bar{y}_m \in \mathbb{N}^k$. Every linear set L_j can directly be translated to a succinct k-track AFT recognising the relation $\{(x_1, \ldots, x_k) \in (\Sigma^*)^k \mid (|x_1|, \ldots, |x_k|) \in L_j\}$, and the union of AFTs be constructed as shown in Section 4.2, resulting in an AFT $\mathcal{R}_{\varphi_{\text{Pres}}}(x_1, \ldots, x_k)$ that is equivalent to $\varphi_{\text{Pres}}(|x_1|, \ldots, |x_k|)$.

6 RATIONAL CONSTRAINTS WITH SYNCHRONISATION PARAMETERS

In order to simplify the decision procedure for SL, which we will present in Section 7, we introduce an enriched syntax of rational constraints. We will then extend the AC decision procedure from Section 5 to the new type of constraints such that it can later be used as a subroutine in our decision procedure of SL. Before giving details, we will outline the main idea behind the extension.

The AC decision procedure expects acyclicity, which prohibits formulae that are, e.g., of the form $(\varphi(x) \wedge \varphi'(y)) \wedge \psi(x, y)$. Indeed, after replacing the inner-most conjunction by an equivalent rational constraint, the formula turns into the conjunction $\mathcal{R}_{\varphi \wedge \varphi'}(x, y) \wedge \mathcal{R}_{\psi}(x, y)$, which is a conjunction of the form $\mathcal{R}(x, y) \wedge \mathcal{S}(x, y)$. In general, satisfiability of such conjunctions is not decidable, and they cannot be expressed as a single AFT since synchronisation of ϵ-moves on multiple tracks is not always possible. However, our example conjunction does not compose two arbitrary AFTs. By its construction, $\mathcal{R}_{\varphi \wedge \varphi'}(x, y)$ actually consists of two disjoint AFT parts. Each of the parts constrains symbols read on one of the two tracks only and is completely oblivious of the other part. Due to this, an AFT equivalent to $\mathcal{R}_{\varphi \wedge \varphi'}(x, y) \wedge \mathcal{R}_{\psi}(x, y)$ can be constructed (let us outline, without so far going into details, that the construction would saturate ϵ-moves for each track of $\mathcal{R}_{\varphi \wedge \varphi'}$ separately). Indeed, the original formula can also be rewritten as $\varphi(x) \wedge (\varphi(y) \wedge \psi(x, y))$, which is AC and can be solved by the algorithm of Section 5.

The idea of exploiting the independence of tracks within a transducer can be taken a step further. The two independent parts do not have to be totally oblivious of each other, as in the case of $\mathcal{R}_{\varphi \wedge \varphi'}$ above, but can communicate in a certain limited way. To define the allowed form of communication and to make the independent communicating parts syntactically explicit within string formulae, we will introduce the notion of synchronisation parameters of AFTs. We will then explain how formulae built from constraints with synchronisation parameters can be transformed into a single rational constraint with parameters by a simple adaptation of the AC algorithm, and how the parameters can be subsequently eliminated, leading to a single standard rational constraint.

Definition 6.1 (AFT with synchronisation parameters). An *AFT with parameters* $\bar{s} = s_1, \ldots, s_n$ is defined as a standard AFT $\mathcal{R} = (V, Q, \Delta, I, F)$ with the difference that the initial and the final formula can talk apart from states about so-called *synchronisation parameters* too. That is, $I, F \subseteq \mathbb{F}_{Q \cup \{\bar{s}\}}$ where I is still positive on states and F is still negative on states, but the synchronisation parameters can appear in I and F both positively as well as negatively. The synchronisation parameters put an additional constraint on accepting runs. A run $\rho = \rho_0 \ldots \rho_m$ over a k-tuple of words $\bar{w}$ is accepting

only if there is a truth assignment $\nu : \{\bar{s}\} \to \mathbb{B}$ of parameters such that $\nu \models I$ and $\nu \models F$. We then say that $\bar{w}$ is *accepted with the parameter assignment* ν.

String formulae can be built on top of AFTs with parameters in the same way as before. We write $\varphi[\bar{s}](\bar{x})$ to denote a string formula that uses AFTs with synchronisation parameters from $\bar{s}$ in its rational constraints. Such a formula is interpreted over a union $\iota \cup \nu$ of an assignment $\iota : \mathrm{var}(\varphi) \to \mathcal{P}(V)^*$ from string variables to strings, as usual, and a parameter assignment $\nu : \{\bar{s}\} \to \mathbb{B}$. An atomic constraint $\mathcal{R}[\bar{s}](\bar{x})$ is satisfied by $\iota \cup \nu$, written $\iota \cup \nu \models \mathcal{R}[\bar{s}](\bar{x})$, if $\mathcal{R}$ accepts $(\iota(x_1), \ldots, \iota(x_{|\bar{x}|}))$ with the parameter assignment ν. Atomic string constraints without parameters are satisfied by $\iota \cup \nu$ iff they are satisfied by ι. The satisfaction $\iota \cup \nu \models \varphi$ of a Boolean combination φ of atomic constraints is then defined as usual.

Notice that within a non-trivial string formula, parameters may be shared among AFTs of several rational constraints. They then not only synchronise initial and final configuration of a single transducer run, but provide the aforementioned limited way of communication among AFTs of the rational constraints within the formula.

*Definition 6.2 (*AC*with synchronisation parameters—*ACsp*).* The definition of AC extends quite straightforwardly to rational constraints with parameters. There is no other change in the definition except for allowing rational constraints to use synchronisation parameters as defined above.

Notice that since we do not consider regular constraints with parameters, constraints with parameters in ACsp formulae are never negated.

The synchronisation parameters allow for an easier transformation of string formulae into AC. For instance, consider a formula of the form $\varphi(x, y) \wedge \psi(x, y)$ where one of the conjuncts, say φ, can be rewritten as $\varphi_1[\bar{s}_1](x) \wedge \varphi_2[\bar{s}_2](y)$. The whole formula can be written as $\varphi_1[\bar{s}_1](x) \wedge (\varphi_2[\bar{s}_2](y) \wedge \psi(x, y))$, which falls into ACsp. An example of such a formula $\varphi(x, y)$, commonly found in the benchmarks we experimented with as presented later on, is a formula saying that $x \circ y$ belongs to a regular language, expressed by an AFA $\mathcal{A}$. This can be easily expressed by a conjunction $\mathcal{R}_1[\bar{s}](x) \wedge \mathcal{R}_2[\bar{s}](y)$ of two unary rational constraints with parameters. Intuitively, the AFTs $\mathcal{R}_1$ and $\mathcal{R}_2$ are two copies of $\mathcal{A}$. $\mathcal{R}_1$ nondeterministicaly chooses a configuration where the prefix of a run of $\mathcal{A}$ reading a word x ends, accepts, and remembers the accepting configuration in parameter values (it will have a parameter per state). $\mathcal{R}_2$ then reads the suffix of x, using the information contained in parameter values to start from the configuration where $\mathcal{R}_1$ ended. We explain this construction in detail in Section 7.

An ACsp formula φ with parameters can be translated into a single, parameter-free, rational constraint and then decided by an AFA language emptiness check described in Section 8. The translation is done in two steps:

(1) A **generalised** AC **algorithm** translates $\varphi(\bar{x})$ to $\mathcal{R}_\varphi[\bar{s}](\bar{x})$.
(2) **Parameter elimination** transforms $\mathcal{R}_\varphi[\bar{s}](\bar{x})$ to a normal rational constraint $\mathcal{R}'_\varphi(\bar{x})$.

Generalised AC *algorithm.* To enable eliminations of conjunctions and disjunctions from ACsp formulae, just a small modification of the procedure from Section 5 is enough. The presence of parameters in the initial and final formulae does not require any special treatment, except that, unlike for states (which are implicitly renamed), it is important that sets of synchronisation parameters stay the same even if they intersect, so that the synchronisation is preserved in the resulting AFT. That is, for $\Box \in \{\wedge, \vee\}$, $\mathcal{R}_\varphi[\bar{r}](\bar{x})$, and $\mathcal{R}_\psi[\bar{s}](\bar{y})$, the constraint $\mathcal{R}_{\varphi\Box\psi}[\bar{t}](\bar{z})$ is created in the same way as described in Section 5, the parameters within the initial and the final formulae of the input AFTs are passed to the AFA construction $\Box$ unchanged, and $\{\bar{t}\} = \{\bar{r}\} \cup \{\bar{s}\}$.

Lemma 6.3. *$\mathcal{R}_\varphi[\bar{r}](\bar{x}) \Box \mathcal{R}_\psi[\bar{s}](\bar{y})$ is equivalent to $\mathcal{R}_{\varphi\Box\psi}[\bar{t}](\bar{z})$.*

Elimination of parameters. The previous steps transform the formula into a single rational constraint with synchronisation parameters. Within such a constraint, every parameter communicates one bit of information between the initial and final configuration of a run. The bit can be encoded by an additional automata state passed from a configuration to a configuration via transitions through the entire run, starting from an initial configuration where the parameter value is decided in accordance with the initial formula, to the final configuration where it is checked against the final formula. A technical complication, however, is that automata transitions are monotonic (positive on states). Hence, they cannot prevent arbitrary states from appearing in target configurations even though their presence is not enforced by the source configuration. For instance, starting from a single state q_1 and executing a transition $\Delta(q_1) = q_2$ can yield a configuration $q_2 \wedge q_3$. The assignment of 0 to a parameter cannot therefore be passed through the run in the form of absence of a single designated state as it can be overwritten anywhere during the run.

To circumvent the above, we use a so-called *two rail encoding* of parameter values: every parameter s is encoded using a pair of value indicator states, the positive value indicator s^+ and the negative value indicator s^-. Addition of unnecessary states into target configurations during a run then cannot cause that a parameter silently changes its value. One of the indicators can still get unnecessarily set, but the other indicator will stay in the configuration too (states can be added into the configurations reached, but cannot be removed). The parameter value thus becomes ambiguous—both s^- and s^+ are present. The negative final formula can exclude all runs which arrive with ambiguous parameters by enforcing that at least one of the indicators is false.

Formally, the parameter elimination replaces a constraint $\mathcal{R}(\bar{x})[\bar{s}]$ with $\mathcal{R} = (W\langle|\bar{x}|\rangle, Q, \Delta, I, F)$ and $|\bar{s}| = n$ by a parameter free constraint $\mathcal{R}'(\bar{x})$ with $\mathcal{R}' = (W\langle|\bar{x}|\rangle, Q', \Delta', I', F')$ where

- $Q' = Q \cup \{s_i^+, s_i^- \mid 1 \leq i \leq n\}$ (parameters are added to Q), and
- $\Delta' = \Delta \cup \{s_i^+ \mapsto s_i^+, s_i^- \mapsto s_i^- \mid 1 \leq i \leq n\}$ (once active value indicators stay active).
- $I' = I^+ \wedge \mathit{Choose}$ where I^+ is a positive formula that arises from I by replacing every negative occurrence of a parameter $\neg s$ by a positive occurrence of its negative indicator s^-, and the positive formula $\mathit{Choose} = \bigwedge_{i=1}^{n} s_i^+ \vee s_i^-$ enforces that every parameter has a value.
- $F' = F^- \wedge \mathit{Disambiguate}$ where F^- is a negative formula that arises from F by replacing every positive occurence of a parameter s by a negative occurrence of its negative indicator $\neg s^-$, and the negative formula $\mathit{Disambiguate} = \bigwedge_{i=1}^{n} \neg s_i^+ \vee \neg s_i^-$ enforces that indicators determine parameter values unambiguously, i.e., at most one indicator per parameter is set.

Lemma 6.4. *$\exists \bar{s} : \mathcal{R}(\bar{x})[\bar{s}]$ is equivalent to $\mathcal{R}'(\bar{x})$.*

7 DECIDING STRAIGHT-LINE FORMULAE

Our algorithm solves string formulae using the DPLL(T) framework [Nieuwenhuis et al. 2004][6], where T is a sound and complete solver for AC and SL. Loosely speaking, DPLL(T) can be construed as a collaboration between a DPLL-based SAT-solver and theory solvers, wherein the input formula is viewed as a Boolean formula by the SAT solver, checked for satisfiability by the SAT-solver, and if satisfiable, theory solvers are invoked to check if the Boolean assignment found by the SAT solver can in fact be realised in the involved theories. The details of the DPLL(T) framework are not so important for our purpose. However, the crucial point is that all queries that a DPLL(T) solver asks a T-theory solver are conjunctions from the CNF of the input formula (or their parts), enabling us to concentrate on solving SL conjunctions only.

Our decision procedure for SL conjunctions transforms the input SL conjunction into an equisatisfiable ACsp formula, which is then decided as discussed in Section 6. The rest of the section is thus devoted to a translation of a positive SL conjunction φ to an ACsp formula. The translation

[6] Also see [Kroening and Strichman 2008] for a gentle introduction to DPLL(T).

internally combines rational constraints and equations into a more general kind of constraints in which rational relations are mixed with concatenations and synchronisation parameters.

Example 7.1. As a running example for the section, we use an SL conjunction that captures the essence of the vulnerability pattern from Example 1.1: A sanitizer is applied on an input string to get rid of symbols c, replacing them by d, hoping that this will prevent a dangerous situation which arises when a symbol d apears in a string somewhere behind c. However, the dangerous situation will not be completely avoided since it is forgotten that the sanitized string will be concatenated with another string that can still contain c.[7]

To formalize the example, assume a bit-vector encoding of an alphabet Σ which contains the symbols c and d. Assume that each $a \in \Sigma$ denotes the conjunction of (negated) bit variables encoding it. As our running example, we will then consider the formula $\varphi : y = \mathcal{R}(x) \wedge z = x \circ y \wedge \mathcal{A}(z)$. The AFT $\mathcal{R} = (W\langle 2\rangle, Q = \{q\}, \Delta = \{q \mapsto q \wedge \neg \mathsf{d}^1 \wedge (\mathsf{c}^1 \rightarrow \mathsf{d}^2) \wedge \bigwedge_{a \in \Sigma \setminus \{\mathsf{c}\}} (a^1 \leftrightarrow a^2))\}, I = q, F = \mathtt{true})$ is a sanitizer that produces y by replacing all occurrences of c in its input string x by d, and it also makes sure that x does not include d. The AFA $\mathcal{A} = (V, Q' = \{r_0, r_1, r_2\}, \Delta', I' = r_0, F' = \neg r_0 \wedge \neg r_1)$ where $\Delta'(r_0) = (r_0 \wedge \neg \mathsf{c}) \vee (r_1 \wedge \mathsf{c})$, $\Delta'(r_1) = (r_1 \wedge \neg \mathsf{d}) \vee (r_2 \wedge \mathsf{d})$, and $\Delta'(r_2) = \mathtt{true}$ is the specification. It checks whether the opening symbol c can be later followed by the closing symbol d in the string z. The formula is satisfiable. □

Definition 7.2 (Mixed constraints). A *mixed constraint* is of the form $x = \mathcal{R}[\bar{s}](y_1 \circ \cdots \circ y_n)$ where $\mathcal{R}$ is a binary AFT, with a concatenation of variables as the right-hand side argument, and $\bar{s}$ is a vector of synchronisation parameters. Such constraint has the expected meaning: it is satisfied by the union $\nu \cup \iota$ of an assignment ι to string variables and an assignment ν to parameters iff $(\iota(x), \iota(y_1) \circ \cdots \circ \iota(y_n))$ is accepted by $\mathcal{R}[\bar{s}]$ with the parameter assignment ν.

All steps of our translation of the input SL formula φ to an ACsp formula preserve the SL fragment, naturally generalised to mixed constraints as follows.

Definition 7.3 (Generalised straight-line conjunction). A conjunction of string constraints is defined to be generalised *straight-line* if it can be written as $\psi \wedge \bigwedge_{i=1}^{m} x_i = F_i$ where ψ is a conjunction over regular and negated regular constraints and each F_i is either of the form $y_1 \circ \cdots \circ y_n$ or $\mathcal{R}[\bar{s}](y_1 \circ \cdots \circ y_n)$ such that it does not contain variables $x_i, \ldots, x_m$.

For simplicity, we assume that φ has gone through two preprocessing steps. First, all negations were eliminated by complementing regular constraints, resulting in a purely positive conjunction. Second, all the—now only positive—regular constraints were replaced by equivalent rational constraints. Particularly, a regular constraint $\mathcal{A}(x)$ is replaced by a rational constraint $x' = \mathcal{R}'(x)$ where x' is a fresh variable and $\mathcal{R}'$ is an AFT with $Rel(\mathcal{R}') = \mathcal{P}(V)^* \times L(\mathcal{A})$. The AFT $\mathcal{R}'$ is created from $\mathcal{A}$ by indexing all propositions in the transition relation by the index 2 of the second track. It is not difficult to see that since x' is fresh, the replacement preserves SL, and also satisfiability, since $P(x) \wedge \psi$ is equivalent to $\exists x' : x' = \mathcal{R}(x) \wedge \psi$ for every ψ.

Example 7.4. In Example 7.1, the preprocessing replaces the conjunct $\mathcal{A}(z)$ by $z' = \mathcal{S}(z)$ where $\mathcal{S}$ is the same as $\mathcal{A}$, except occurrences of bit-vector variables in Δ' are indexed by 2 since z will be read on its second track. We obtain $\varphi'_0 : y = \mathcal{R}(x) \wedge z = x \circ y \wedge z' = \mathcal{S}(z)$ where z' is free. □

Due to the preprocessing, we are starting with a formula φ'_0 in the form of an SL conjunction of rational constraints and equations. The translation to ACsp will be carried out in the following three steps, which will be detailed in the rest of the section:

[7] In reality, where one undesirably concatenates a string `command('...` with some string `...');attack();` the situation is, of course, more complex and sanitization is more sophisticated. However, having a real-life example, such as those used in our experiments, as a running example would be too complex to understand.

(1) **Substitution** transforms φ_0' to a conjunction φ_1 of mixed constraints.
(2) **Splitting** transforms φ_1 to a conjunction φ_2 of rational constraints with parameters.
(3) **Ordering** transforms φ_2 to an AC conjunction φ_3 with parameters.

Substitution. Equations in φ_0' are combined with rational constraints into mixed constraints by a straightforward substitution. In one substitution step, a conjunction $x = y_1 \circ \cdots \circ y_n \wedge \psi$ is replaced by $\psi[y_1 \circ \cdots \circ y_n/x]$ where all occurrences of x are replaced by $y_1 \circ \cdots \circ y_n$. The substitution preserves the generalised straight-line fragment.

Lemma 7.5. *If $x = y_1 \circ \cdots \circ y_n \wedge \psi$ is* SL*, then $\psi[y_1 \circ \cdots \circ y_n/x]$ is equisatisfiable and* SL.

The substitution steps are iterated eagerly in an arbitrary order until there are no equations. Every substitution step obviously decreases the number of equations, so the iterative process terminates after a finitely many steps with an equation-free SL conjunction of mixed constraints φ_1.

Example 7.6. The substitution eliminates the equation $z = x \circ y$ in φ_0' from Example 7.4, transforming it to $\varphi_1 : y = \mathcal{R}(x) \wedge u = \mathcal{S}(x \circ y)$. □

Splitting. We will now explain how synchronisation parameters are used to eliminate concatenation within mixed constraints. The operation of *binary splitting* applied to an SL conjunction of mixed constraints, $\varphi : x = \mathcal{R}(y_1 \circ \cdots \circ y_m \ \circ \ z_1 \circ \cdots \circ z_n)[\bar{s}] \wedge \psi$, where $\mathcal{R} = (W\langle 2\rangle, Q, \Delta, I, F)$ and $Q = \{q_1, \ldots, q_l\}$ splits the mixed constraint and substitutes x by a concatenation of fresh variables $x_1 \circ x_2$ in ψ. That is, it outputs the conjunction $\varphi' : \zeta \wedge \psi[x_1 \circ x_2/x]$ of mixed constraints, where the rational constraint was split into the following conjunction ζ of two constraints:

$$\zeta : x_1 = \mathcal{R}_1(y_1 \circ \cdots \circ y_m)[\bar{s}, \bar{t}] \wedge x_2 = \mathcal{R}_2(z_1 \circ \cdots \circ z_n)[\bar{s}, \bar{t}]$$

The vector $\bar{t}$ consists of l fresh parameters, x_1 and x_2 are fresh string variables, and each AFT with parameters $\mathcal{R}_i = (W\langle 2\rangle, Q, \Delta, I_i, F_i)$, $i \in \{1, 2\}$, is derived from $\mathcal{R}$ by choosing initial/final formulae:

$$I_1 = I, \quad F_1 = \bigwedge_{i=1}^{l} q_i \to t_i, \qquad I_2 = \bigwedge_{i=1}^{l} t_i \to q_i, \quad F_2 = F\,.$$

Intuitively, each run ρ of $\mathcal{R}$ is split into a run ρ_1 of $\mathcal{R}_1$, which corresponds to the first part of ρ in which $y_1 \circ \cdots \circ y_m$ is read along with a prefix x_1 of x, and a run ρ_2 of $\mathcal{R}_2$, which corresponds to the part of ρ in which $z_1 \circ \cdots \circ z_n$ is read along with the suffix x_2 of x. Using the new synchronisation parameters $\bar{t}$, the formulae F_1 and I_2 ensure that the run ρ_1 of $\mathcal{R}_1$ must indeed start in the states in which the run ρ_2 of $\mathcal{R}_2$ ended, that is, the original run ρ of $\mathcal{R}$ can be reconstructed by connecting ρ_1 and ρ_2. Every occurrence of x in ψ is replaced by the concatenation $x_1 \circ x_2$.

Lemma 7.7. *In the above, φ is equivalent to $\exists x_1 x_2 \bar{t} : \varphi'$.*

The resulting formula φ' is hence equisatisfiable to the original φ. Moreover, φ' is still generalised SL—the two new constraints defining x_1 and x_2 can be placed at the position of the original constraint defining x that was split, and the substitution $[x_1 \circ x_2/x]$ in the rest of the formula only applies to the right-hand sides of constraints (since x can be defined only once).

Lemma 7.8. *If φ is an* SL *conjunction of mixed constraints, then so is φ'.*

Moreover, by applying binary splitting steps eagerly in an arbitrary order on φ_1, we are guaranteed that all concatenations will be eliminated after a finite number of steps, thus arriving at the SL conjunction of rational constraints with parameters φ_2. The termination argument relies on the straight-line restriction. Although it cannot be simply said that every step reduces the number of concatenations because the substitution $x_1 \circ x_2$ introduces new ones, the new concatenations $x_1 \circ x_2$ are introduced only into constraints defining variables that are higher in the straight-line

ordering than x. It is therefore possible to define a well-founded (integer) measure on the formulae that decreases with every application of the binary splitting steps.

Lemma 7.9. *All concatenations in the* SL *conjunction of mixed constraints φ_1 will be eliminated after a finite number of binary splitting steps.*

We note that our implementation actually uses a slightly more efficient n-ary splitting instead of the described binary. It splits a mixed constraint in one step into the number of conjuncts equal to the length of the concatenation in its right-hand side. We present the simpler binary variant, which eventually achieves the same effect.

Example 7.10. The formula from Example 7.6 would be transformed into $\varphi_2 : y = \mathcal{R}(x) \wedge u_1 = \mathcal{S}_1[\bar{s}](x) \wedge \mathcal{S}_2[\bar{s}](y)$ where $\mathcal{S}_1, \mathcal{S}_2$ are as $\mathcal{S}$ up to that $\mathcal{S}_1$ has the final formula $I' \wedge \bigwedge_{i=0}^{2}(r_i \to s_0)$ and $\mathcal{S}_2$ has the final formula $F' \wedge \bigwedge_{i=0}^{2}(s_i \to r_i)$. Notice that $u_1 = \mathcal{S}_1[\bar{s}](x) \wedge u_2 = \mathcal{S}_2[\bar{s}](y)$ still enforce that $x \circ y$ has c eventually followed by d. The parameters remember where $\mathcal{S}_1$ ended its run and force $\mathcal{R}_2$ to continue from the same state. □

Reordering modulo associativity. Substitution and splitting transform φ_0 to a straight-line conjunction φ_2 of rational constraints with parameters. Before delegating it to the ACsp formulae solver, it must be reorganized modulo associativity to achieve a structure satisfying the definition of AC. One way of achieving this is to order the formula into a conjunction $\bigwedge_{i=1}^{m} x_i = \mathcal{R}[\bar{s}^i](y_i)$ satisfying the condition in the definition of SL (the definition of SL only requires that the formula *can* be assumed). An simple way is discussed in [Lin and Barceló 2016]. It consists of drawing the dependency graph of φ, a directed graph with the variables $\mathrm{var}(\varphi)$ as vertices which has an edge $x \to y$ if and only if φ contains a conjunct $x = \mathcal{R}(y)$. Due to the straight-line restriction, the graph must be acyclic. The ordering of variables can be then obtained as a topological sort of the graphs vertices, which is computable in linear time (e.g. [Cormen et al. 2009], for instance by a depth-first traversal). The final acyclic formula φ_3 then arises when letting $\bigwedge_{i=1}^{m}$ associate from the right:

$$\varphi_3 : (x_1 = \mathcal{R}_1(y_1) \wedge (x_2 = \mathcal{R}_2(y_2) \wedge (\ldots \wedge (x_{m-1} = \mathcal{R}_{m-1}(y_{m-1}) \wedge x_m = \mathcal{R}_m(y_m)) \ldots))).$$

To see that φ_3 is indeed ACsp, observe that every conjunctive sub-formula is of the form $(\bigwedge_{i<k} x_i = \mathcal{R}_i(y_i)) \wedge x_k = \mathcal{R}_k(y_k)$ where x_k is by the definition of SL not present in the left conjunct. The left and right conjuncts can therefore share at most one variable, y_k.

Theorem 7.11. *The formula φ_3 obtained by substitution, splitting, and reordering from φ_0 is equisatisfiable and acyclic.*

Example 7.12. The ACsp formula $\varphi_3 : y = \mathcal{R}(x) \wedge u_1 = \mathcal{S}_1[\bar{s}](x)) \wedge \mathcal{S}_2[\bar{s}](y)$ would be the final result of the SL to ACsp translation. Let us use φ_3 to also briefly illustrate the decision procedure for ACsp of Section 6. The first step is the transformation to a single rational constraint with parameters by induction over formula structure. This will produce $\mathcal{R}'[\bar{s}](x, y, z)$ with states and transitions consisting of those in $\mathcal{R}$, $\mathcal{S}_1$ with indexes of alphabet bits incremented by one (y, and z are now not the first and the second, but the second and the third track), and a copy $\mathcal{S}_2'$ of $\mathcal{S}_2$ with states replaced by their primed variant (so that they are disjoint from that of $\mathcal{S}_1$) and also incremented indexes of alphabet bits. The initial and final configuration will be the conjunctions of those of $\mathcal{R}, \mathcal{S}_1$ and $\mathcal{S}_2'$. The last step, eliminating of parameters, will lead to the addition of positive and negative indicator states for parameters $\bar{s} = s_1, s_2, s_3$ with the universal self-loops and the update of the initial and final formula as in Section 6. The rest is solved by the emptiness check discussed in Section 8. Notice the small size of the resulting AFT. Compared to the original formula from Example 7.1, it contains only one additional copy of $\mathcal{A}$ (the $\mathcal{S}_2'$), the six additional parameter indicator states with self-loops and the initial and final condition on the parameter indicators. □

A note on the algorithm of [Lin and Barceló 2016]. We will now comment on the differences of our algorithm for deciding SL from the earlier algorithm of [Lin and Barceló 2016]. It combines reasoning on the level NFAs and nondeterministic transducers, utilising classical automata theoretic techniques, with a technique for eliminating concatenation by enumerative automata splitting. It first turns and SL formula into a pure AC formula and then uses the AC decision procedure.

An obvious advantage of our decision procedure described in Section 5 is the use of succinct AFA. As opposed to the worst case exponentially larger NFA, it produces an AFA of a linear size (unless the original formula contains negated regular constraints represented as general AFA. See Section 5 for a detailed discussion). Let us also emphasize the advantages of our algorithm in the first phase, translation of SL to ACsp. Similarly as in the case of deciding AC, the main advantage of our algorithm is that, while [Lin and Barceló 2016] only works with NFTs, we propose ways of utilising the power of alternation and succinct transition encoding.

We will illustrate the difference on an example. The concatenation in the conjunction $x = y \circ z \wedge w = \mathcal{R}(x)$ would in [Lin and Barceló 2016] be done by enumerative splitting. It replaces the conjunction by the disjunction $\bigvee_{q \in Q} w_1 = \mathcal{R}_q(y) \wedge w_2 = {}_q\mathcal{R}(z)$. The Q in the disjunction is the set of states of the (nondeterministic) transducer $\mathcal{R}$, $\mathcal{R}_q$ is the same as the NFT $\mathcal{R}$ up to that the final state is q, and ${}_q\mathcal{R}$ the same as $\mathcal{R}$ up to that the initial state is q. Intuitively, the run of $\mathcal{R}$ is explicitly separated into the part in which y is read along the prefix w_1 of w, and the suffix in which z is red along the suffix w_2 of w. The variable w would be replaced by $w_1 \circ w_2$ in the rest of the formula. The disjunction enumerates all admissible intermediate states $q \in Q$ a run of $\mathcal{R}$ can cross, and for each of them, it constructs two copies of $\mathcal{R}$. This makes the cost of the transformation quadratic in the number of states of the NFT $\mathcal{R}$. A straightforward generalisation to our setting in which $\mathcal{R}$ is an AFT is possible: The disjunction would have to list, instead of possible intermediate states $q \in Q$, all possible intermediate configurations $C \subseteq Q$ a run of the AFA $\mathcal{R}$ can cross, thus increasing the quadratic blow-up of the nondeterministic case to an exponential (due to the enumerative nature of splitting, the size is without any optimisation bounded by an exponential even from below).

Our splitting algorithm utilises succinctness of alternation to reduce the cost of enumerative AFA splitting from exponential space (or quadratic in the case of NFAs) to linear. The smaller size of the resulting representation is payed for by a more complex alternating structure of the resulting rational constraints. The worst case complexity of the satisfiability procedure thus remains essentially the same. However, deferring most of the complexity to the last phase of the decision procedure, AFA emptiness checking, allows to circumvent the potential blow-up by means of modern model checking algorithms and heuristics and achieve much better scalability in practice.

8 MODEL CHECKING FOR AFA LANGUAGE EMPTINESS

In order to check unsatisfiability of a string formula using our translation to AFTs, it is necessary to show that the resulting AFT does not accept any word, i.e., that the recognised language is empty. The constructed AFTs are succinct, but tend to be quite complex: a naïve algorithm that would translate AFTs to NFAs using an explicit subset construction, followed by systematic state-space exploration, is therefore unlikely to scale to realistic string problems. We discuss how the problem of AFT emptiness can instead be reduced (in linear time and space) to reachability in a Boolean transition system, in a way similar to [Cox and Leasure 2017; Gange et al. 2013; Wang et al. 2016]. Our translation is also inspired by the use of model checkers to determinise NFAs in [Tabakov and Vardi 2005], by a translation to sequential circuits that corresponds to symbolic subset construction. We use a similar implicit construction to map AFAs and AFTs to NFAs.

As an efficiency aspect of the construction for AFAs, we observe that it is enough to work with *minimal* sets of states, thanks to the monotonicity properties of AFAs (the fact that initial formulae and transition formulae are positive in the state variables, and final formulae are negative). This

gives rise to three different versions: a direct translation that does not enforce minimality at all; an *intensionally-minimal* translation that only considers minimal sets by virtue of additional Boolean constraints; and a *deterministic* translation that resolves nondeterminism through additional system inputs, but does not ensure fully-minimal state sets.

8.1 Direct Translation to Transition Systems

To simplify the presentation of our translation to a Boolean transition system, we focus on the case of AFAs $\mathcal{A} = (V_n, Q, \Delta, I, F)$ over a single track of bit-vectors of length $n+1$. The translation directly generalises to k-track AFTs, and to AFTs with epsilon characters, by simply choosing n sufficiently large to cover the bits of all tracks.

We adopt a standard Boolean transition system view on the execution of the AFA $\mathcal{A}$ (e.g., [Clarke et al. 1999]). If $\mathcal{A}$ has $m = |Q|$ automaton states, then $\mathcal{A}$ can be interpreted as a (symbolically described) transition system $T^{di}_{\mathcal{A}} = (\mathbb{B}^m, \mathit{Init}^{di}, \mathit{Trans}^{di})$. The transition system has state space $\mathbb{B}^m$, i.e., a system state is a bit-vector $\bar{q} = \langle q_0, \ldots, q_{m-1} \rangle$ of length m identifying the active states in Q. The initial states of the system are defined by $\mathit{Init}^{di}[\bar{q}] = I$, the same positive Boolean formula as in $\mathcal{A}$. The transition relation Trans^{di} of the system is a Boolean formula over two copies $\bar{q}, \bar{q}'$ of the state variables, encoding that for each active pre-state q_i in $\bar{q}$ the formula $\Delta(q_i)$ has to be satisfied by the post-state $\bar{q}'$. Input variables $V_n = \{x_0, \ldots, x_n\}$ are existentially quantified in the transition formula, expressing that all AFA transitions have to agree on the letter to be read:

$$\mathit{Trans}^{di}[\bar{q}, \bar{q}'] \;=\; \exists v_0, \ldots, v_n : \bigwedge_{i=0}^{m-1} q_i \rightarrow \Delta(q_i)[\bar{q}/\bar{q}'] \tag{1}$$

To examine emptiness of $\mathcal{A}$, it has to be checked whether $T^{di}_{\mathcal{A}}$ can reach any state in the target set $\mathit{Final}^{di}[\bar{q}] = F$, i.e., in the set described by the negative final formula F of $\mathcal{A}$. Since is well-known that reachability in transition systems is a PSPACE-complete problem [Clarke et al. 1999], this directly establishes that fragment AC is in PSPACE (Corollary 5.4).

Lemma 8.1. *The language $L(\mathcal{A})$ recognised by the AFA $\mathcal{A}$ is empty if and only if $T^{di}_{\mathcal{A}}$ cannot reach a configuration in $\mathit{Final}^{di}[\bar{q}]$.*

In practice, this means that emptiness of $L(\mathcal{A})$ can be decided using a wide range of readily available, highly optimised model checkers from the hardware verification field, utilising methods such as k-induction [Sheeran et al. 2000], Craig interpolation [McMillan 2003], or IC3/PDR [Bradley 2012]. In our implementation, we represent $T^{di}_{\mathcal{A}}$ in the AIGER format [Biere et al. 2017], and then apply nuXmv [Cavada et al. 2014] and ABC [Brayton and Mishchenko 2010].

The encoding $T^{di}_{\mathcal{A}}$ leaves room for optimisation, however, as it does not fully exploit the structure of AFAs and introduces more transitions than strictly necessary. In (1), we can observe that if $\mathit{Trans}^{di}[\bar{q}, \bar{q}']$ is satisfied for some $\bar{q}, \bar{q}'$, then it will also be satisfied for every post-state $\bar{q}'' \geq \bar{q}'$, writing $\bar{p} \preceq \bar{q}$ for the point-wise order on bit-vectors $\bar{p}, \bar{q} \in \mathbb{B}^m$ (i.e., $\bar{p} \preceq \bar{q}$ if p_i implies q_i for every $i \in \{0, \ldots, m-1\}$). This is due to the positiveness (or *monotonicity*) of the transition formulae $\Delta(q_i)$. Similarly, since the initial formula I of an AFA is positive, initially more states than necessary might be activated. Because the final formula F is negative, and since redundant active states can only impose additional restrictions on the possible runs of an AFA, such redundant states can never lead to more words being accepted.

More formally, we can observe that the transition system $T^{di}_{\mathcal{A}}$ is *well-structured* [Finkel 1987], which means that the state space $\mathbb{B}^m$ can be equipped with a well-quasi-order $\leq$ such that whenever $\mathit{Trans}^{di}[\bar{q}, \bar{q}']$ and $\bar{q} \leq \bar{p}$, then there is some state $\bar{p}'$ with $\bar{q}' \leq \bar{p}'$ and $\mathit{Trans}^{di}[\bar{p}, \bar{p}']$. In our case, $\leq$ is

the inverse point-wise order $\geq$ on bit-vectors;[8] intuitively, deactivating AFA states can only enable more transitions. Since the set $Final^{di}[\bar{q}]$ is upward-closed with respect to $\leq$ (downward-closed with respect to $\preceq$), the theory on well-structured transition systems tells us that it is enough to consider transitions to $\leq$-maximal states (or $\preceq$-minimal states) of the transition system when checking reachability of $Final^{di}[\bar{q}]$. In forward-exploration of the reachable states of $T^{di}_{\mathcal{A}}$, the non-redundant states to be considered form an anti-chain. This can be exploited by defining tailor-made exploration algorithms [Doyen and Raskin 2010; Kloos et al. 2013], or, as done in the next sections, by modifying the transition system to only include non-redundant transitions.

8.2 Intensionally-Minimal Translation

We introduce several restricted versions of the transition system $T^{di}_{\mathcal{A}}$, by removing transitions to non-minimal states. The strongest transition system $T^{\min}_{\mathcal{A}} = (\mathbb{B}^m, Init^{\min}, Trans^{\min})$ obtained in this way can abstractly be defined as:

$$Init^{\min}[\bar{q}] \;=\; Init^{di}[\bar{q}] \wedge \forall \bar{p} \prec \bar{q}.\ \neg Init^{di}[\bar{p}] \tag{2}$$

$$Trans^{\min}[\bar{q}, \bar{q}'] \;=\; Trans^{di}[\bar{q}, \bar{q}'] \wedge \forall \bar{p} \prec \bar{q}'.\ \neg Trans^{di}[\bar{q}, \bar{p}] \tag{3}$$

That means, $Init^{\min}$ and $Trans^{\min}$ are defined to only retain the $\preceq$-minimal states. Computing $Init^{\min}$ and $Trans^{\min}$ corresponds to the logical problem of *circumscription* [McCarthy 1980], i.e., the computation of the set of minimal models of a formula. Circumscription is in general computationally hard, and its precise complexity still open in many cases; in (2) and (3), note that eliminating the universal quantifiers (as well as the universal quantifiers introduced by negation of $Trans^{di}$) might lead to an exponential increase in formula size, so that $T^{\min}_{\mathcal{A}}$ does not directly appear useful as input to a model checker.

We can derive a more practical, but weaker system $T^{\mathrm{im}}_{\mathcal{A}} = (\mathbb{B}^m, Init^{\mathrm{im}}, Trans^{\mathrm{im}})$ by only minimising post-states in $Trans^{\mathrm{im}}$ with respect to the same input letter V_n:

$$\begin{aligned} Init^{\mathrm{im}}[\bar{q}] \;&=\; Init^{\min}[\bar{q}] \\ Trans^{\mathrm{im}}[\bar{q}, \bar{q}'] \;&=\; \exists V_n.\ \Big(Trans[\bar{q}, \bar{q}', V_n] \wedge \forall \bar{p} \prec \bar{q}'.\ \neg Trans[\bar{q}, \bar{p}, V_n] \Big) \\ \text{with} \quad Trans[\bar{q}, \bar{q}', V_n] \;&=\; \bigwedge_{i=0}^{m-1} q_i \rightarrow \Delta(q_i)[\bar{q}/\bar{q}'] \end{aligned}$$

The formulae still contain universal quantifiers $\forall \bar{p}$, but it turns out that the quantifiers can now be eliminated with only polynomial effort, due to the fact that $\bar{p}$ only occurs negatively in the scope of the quantifier. Indeed, whenever $\varphi[\bar{q}]$ is a formula that is positive in $\bar{q}$, and $\varphi[\bar{q}]$ holds for assignments $\bar{q}_1, \bar{q}_3 \in \mathbb{B}^m$ with $\bar{q}_1 \preceq \bar{q}_3$, then $\varphi[\bar{q}]$ will also hold for any assignment $\bar{q}_2 \in \mathbb{B}^m$ with $\bar{q}_1 \preceq \bar{q}_2 \preceq \bar{q}_3$ due to monotonicity. This implies that a satisfying assignment $\bar{q}_1 \in \mathbb{B}^m$ is $\preceq$-minimal if no single bit in $\bar{q}_1$ can be switched from 1 to 0 without violating $\varphi[\bar{q}]$. More formally, $\varphi[\bar{q}] \wedge \neg\exists \bar{p} \prec \bar{q}.\ \varphi[\bar{p}]$ is equivalent to $\varphi[\bar{q}] \wedge \bigwedge_{i=0}^{m-1} q_i \rightarrow \neg\varphi[\bar{q}][q_i/\texttt{false}]$, where we write $\varphi[q_i/\texttt{false}]$ for the result of substituting q_i with `false` in φ.

[8] Since the state space $\mathbb{B}^m$ of $T^{di}_{\mathcal{A}}$ is finite, the "well-" part is trivial.

The corresponding, purely existential representation of $Init^{\text{im}}$ and $Trans^{\text{im}}$ is:

$$Init^{\text{im}}[\bar{q}] \;\equiv\; Init^{di}[\bar{q}] \wedge \bigwedge_{i=0}^{m-1} q_i \rightarrow \neg Init^{di}[\bar{q}][q_i/\texttt{false}] \tag{4}$$

$$Trans^{\text{im}}[\bar{q},\bar{q}'] \;\equiv\; \exists V_n.\, \left(Trans[\bar{q},\bar{q}',V_n] \wedge \bigwedge_{i=0}^{m-1} q'_i \rightarrow \neg Trans[\bar{q},\bar{q}',V_n][q'_i/\texttt{false}]\right) \tag{5}$$

The representation is quadratic in size of the original formulae $Init^{di}$, $Trans^{di}$, but the formulae can in practice be reduced drastically by sharing of common sub-formulae, since the m copies of $Init^{di}[\bar{q}][q_i/\texttt{false}]$ and $Trans[\bar{q},\bar{q}',V_n][q'_i/\texttt{false}]$ tend to be almost identical.

Lemma 8.2. *The following statements are equivalent:*

(1) $T^{di}_{\mathcal{A}}$ can reach a configuration in $Final^{di}[\bar{q}]$;
(2) $T^{min}_{\mathcal{A}}$ can reach a configuration in $Final^{di}[\bar{q}]$;
(3) $T^{im}_{\mathcal{A}}$ can reach a configuration in $Final^{di}[\bar{q}]$.

Example 8.3. To illustrate the $T^{im}_{\mathcal{A}}$ encoding, we consider an AFA $\mathcal{A}$ that accepts the language $\{xwy \mid |xwy| = 2k, k \geq 1, x \in \{a,b\}, y \in \{c,d\}\}$ using the encoding of the alphabet $\Sigma = \{a,b,c,d\}$ from Example 4.1. We let $\mathcal{A} = (\{v_0,v_1\},\{q_0,q_1,q_2,q_3,q_4\},\Delta,I,F)$ where $I = q_0$, $F = \neg q_0 \wedge \neg q_1 \wedge \neg q_3$ (i.e., the accepting states are q_2 and q_4), and Δ is defined as $\Delta(q_0) = \neg v_1 \wedge q_1 \wedge q_3$, $\Delta(q_1) = q_2$, $\Delta(q_2) = q_1$, $\Delta(q_3) = q_3 \;\vee\; (v_1 \wedge q_4)$, and $\Delta(q_4) = \texttt{false}$.

The direct transition system representation is $T^{di}_{\mathcal{A}} = (\mathbb{B}^5, Init^{di}, Trans^{di})$, defined by:

$$Init^{di}[\bar{q}] \;=\; q_0, \qquad Trans^{di}[\bar{q},\bar{q}'] \;=\; \exists v_0,v_1.\, \underbrace{\begin{pmatrix}(q_0 \rightarrow \neg v_1 \wedge q'_1 \wedge q'_3) \wedge \\ (q_1 \rightarrow q'_2) \wedge \\ (q_2 \rightarrow q'_1) \wedge \\ (q_3 \rightarrow q'_3 \;\vee\; (v_1 \wedge q'_4)) \wedge \\ (q_4 \rightarrow \texttt{false})\end{pmatrix}}_{Trans[\bar{q},\bar{q}',V_n]}$$

The intensionally-minimal translation $T^{im}_{\mathcal{A}}$ can be derived from $T^{di}_{\mathcal{A}}$ by conjoining the restrictions in (4) and (5) ($Trans^{im}[\bar{q},\bar{q}']$ is shown in simplified form for sake of presentation):

$$Init^{im}[\bar{q}] \;=\; q_0 \wedge (q_0 \rightarrow \neg\texttt{false}) \wedge \bigwedge_{i=1}^{4} (q_i \rightarrow \neg q_0) \;\equiv\; q_0 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_3 \wedge \neg q_4$$

$$Trans^{im}[\bar{q},\bar{q}'] \;\equiv\; \exists v_0,v_1.\, \begin{pmatrix}Trans[\bar{q},\bar{q}',V_n] \wedge \neg q'_0 \wedge (q'_1 \rightarrow q_0 \vee q_2) \wedge (q'_2 \rightarrow q_1) \wedge \\ (q'_3 \rightarrow q_0 \vee (q_3 \wedge \neg(v_1 \wedge q'_4))) \wedge (q'_4 \rightarrow q_3 \wedge \neg q'_3)\end{pmatrix}$$ □

8.3 Deterministic Translation

We introduce a further encoding of $\mathcal{A}$ as a transition system that is more compact than (4), (5), but does not always ensure fully-minimal state sets. The main idea of the encoding is that a conjunctive transition formula $\Delta(q_1) = q_2 \wedge q_3$, assuming that q_2, q_3 do not occur in any other transition formula $\Delta(q_i)$, can be interpreted as a set of deterministic updates $q'_2 = q_1; q'_3 = q_1$. For state variables that occur in multiple transition formulae, the right-hand side of the update turns into a disjunction. Disjunctions in transition formulae represent nondeterministic updates that can be resolved using additional Boolean flags. The resulting transition system is deterministic, as transitions are uniquely determined by the pre-state and variables representing system inputs.

Example 8.4. We illustrate the encoding $T^{det}_{\mathcal{A}} = (\mathbb{B}^m, Init^{det}, Trans^{det})$ using the AFA from Example 8.3. While the initial states $Init^{det}[\bar{q}]$ coincide with $Init^{im}[\bar{q}]$ in Example 8.3, the transition relation $Trans^{det}[\bar{q}, \bar{q}']$ now consists of two parts: a deterministic assignment of the post-state $\bar{q}'$ in terms of the pre-state $\bar{q}$, together with an auxiliary variable h_3 that determines which branch of $\Delta(q_3)$ is taken; and a conjunct that ensures that value of h_3 is consistent with the inputs V_n. The resulting $Trans^{det}[\bar{q}, \bar{q}']$ is (in this example) equivalent to $Trans^{im}[\bar{q}, \bar{q}']$:

$$
\begin{aligned}
Init^{det}[\bar{q}] \;&=\; q_0 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_3 \wedge \neg q_4 \\
Trans^{det}[\bar{q}, \bar{q}'] \;&\equiv\; \exists h_3.\, \begin{pmatrix} (q_0' \leftrightarrow \texttt{false}) \wedge \\ (q_1' \leftrightarrow q_0 \vee q_2) \wedge \\ (q_2' \leftrightarrow q_1) \wedge \\ (q_3' \leftrightarrow q_0 \vee q_3 \wedge h_3) \wedge \\ (q_4' \leftrightarrow q_3 \wedge \neg h_3) \end{pmatrix} \wedge \exists v_0, v_1.\, \begin{pmatrix} (q_0 \rightarrow \neg v_1) \wedge \\ (q_3 \wedge \neg h_3 \rightarrow v_1) \wedge \\ (q_4 \rightarrow \texttt{false}) \end{pmatrix}
\end{aligned}
$$

□

To define the encoding formally, we make the simplifying assumption that there is a unique initial state q_0, i.e., $I = q_0$, and that all transition formulae $\Delta(q_i)$ are in negation normal form (i.e., in particular state variables in $\Delta(q_i)$ do not occur underneath negation). Both assumption can be established by simple transformation of $\mathcal{A}$. The transition system $T^{det}_{\mathcal{A}} = (\mathbb{B}^m, Init^{det}, Trans^{det})$ is:

$$
\begin{aligned}
Init^{\mathrm{det}}[\bar{q}] \;&=\; q_0 \wedge \bigwedge_{i=1}^{m-1} \neg q_i \\
Trans^{\mathrm{det}}[\bar{q}, \bar{q}'] \;&=\; \exists H.\, \left(\left(\bigwedge_{i=0}^{m-1} q_i' \leftrightarrow \mathit{NewState}(q_i) \right) \wedge \exists V_n.\, \left(\bigwedge_{i=0}^{m-1} q_i \rightarrow \mathit{InputInv}(\Delta(q_i), i) \right) \right)
\end{aligned}
$$

The transition relation $Trans^{\mathrm{det}}$ consists of two main parts: the state updates, which assert that every post-state variable q_i' is set to an update formula $\mathit{NewState}(q_i)$; and an input invariant asserting that the letters that are read are consistent with the transition taken. To determinise disjunctions in transition formulae $\Delta(q_i)$, a set H of additional Boolean variables h_l (uniquely indexed by a position sequence $l \in \mathbb{Z}^*$) is introduced, and existentially quantified in $Trans^{\mathrm{det}}$.

The update formulae $\mathit{NewState}(q_i)$ are defined as a disjunction of assignments extracted from the transition formulae $\Delta(q_j)$,

$$
\mathit{NewState}(q_i) \;=\; \bigvee \{ \varphi \mid \text{there is } j \in \{0, \ldots, m-1\} \text{ such that } \langle q_i, \varphi \rangle \in \mathit{StateAsgn}(\Delta(q_j), j, q_j) \}
$$

where each $\mathit{StateAsgn}(\Delta(q_j), j, q_j)$ represents the set of asserted state variables q_i in $\Delta(q_j)$, together with guards φ for the case that q_i occurs underneath disjunctions. The set is recursively defined (on formulae in NNF) as follows:

$$
\begin{aligned}
\mathit{StateAsgn}(\varphi_1 \wedge \varphi_2, l, g) \;&=\; \mathit{StateAsgn}(\varphi_1, l, g) \cup \mathit{StateAsgn}(\varphi_2, l, g) \\
\mathit{StateAsgn}(\varphi_1 \vee \varphi_2, l, g) \;&=\; \mathit{StateAsgn}(\varphi_1,\, l.1,\, g \wedge h_l) \cup \mathit{StateAsgn}(\varphi_2,\, l.2,\, g \wedge \neg h_l) \\
\mathit{StateAsgn}(q_i, l, g) \;&=\; \{\langle q_i, g \rangle\} \\
\mathit{StateAsgn}(\phi, l, g) \;&=\; \emptyset \qquad\qquad \text{(for any other } \phi\text{)} .
\end{aligned}
$$

In particular, the case for disjunctions $\varphi_1 \vee \varphi_2$ introduces a fresh variable $h_l \in H$ (indexed by the position l of the disjunction) that controls which branch is taken. Input variables $v_i \in V_n$ are ignored in the updates.

The input invariants $\mathit{InputInv}(\Delta(q_i), i)$ are similarly defined recursively, and include the same auxiliary variables $h_l \in H$, but ensure input consistency:

$$\begin{aligned}
&\mathit{InputInv}(\varphi_1 \wedge \varphi_2, l) \;=\; \mathit{InputInv}(\varphi_1, l) \wedge \mathit{InputInv}(\varphi_2, l)\\
&\mathit{InputInv}(\varphi_1 \vee \varphi_2, l) \;=\; \big(h_l \rightarrow \mathit{InputInv}(\varphi_1, l.1)\big) \wedge \big(\neg h_l \rightarrow \mathit{InputInv}(\varphi_2, l.2)\big)\\
&\mathit{InputInv}(v_i, l) \;=\; v_i, \quad \mathit{InputInv}(\neg v_i, l) \;=\; \neg v_i, \quad \mathit{InputInv}(q_i, l) \;=\; \mathtt{true}, \quad \mathit{InputInv}(\phi, l) \;=\; \phi\,.
\end{aligned}$$

9 IMPLEMENTATION AND EXPERIMENTS

We have implemented our method for deciding conjunctive AC and SL formulae as a solver called Sloth (**S**tring **LO**gic **TH**eory solver), extending the Princess SMT solver [Rümmer 2008]. The solver Sloth can be obtained from https://github.com/uuverifiers/sloth/wiki. Hence, Princess provides us with infrastructure such as an implementation of DPLL(T) or facilities for reading input formulae in the SMT-LIBv2 format [Barrett et al. 2010]. Like Princess, Sloth was implemented in Scala. We present results from several settings of our tool featuring different optimizations.

- Sloth-1 The basic version of Sloth, denoted as Sloth-1 below, uses the direct translation of the AFA emptiness problem to checking reachability in transition systems described in Section 8.1. Then, it employs the nuXmv model checker [Cavada et al. 2014] to solve the reachability problem via the IC3 algorithm [Bradley 2012], based on property-directed state space approximation. Further, we have implemented five optimizations/variants of the basic solver: four of them are described below, the last one at the end of the section.
- Sloth-2 Our first optimization, implemented in Sloth-2, is rather simple: We assume working with strings over an alphabet Σ and look for equations of the form $x = a_0 \circ y_1 \circ a_1 \ldots \circ y_n \circ a_n$ where $n \geq 1$, $\forall 0 \leq i \leq n : a_i \in \Sigma^*$ (i.e., a_i are constant strings), and, for every $1 \leq j \leq n$, y_j is a free string variable not used in any other constraint. The optimization replaces such constraints by a regular constraint $(a_0 \circ \Sigma^* \circ a_1 \ldots \circ \Sigma^* \circ a_n)(x)$. This step allows us to avoid many split operations. The optimization is motivated by a frequent appearance of constraints of the given kind in some of the considered benchmarks. As shown by our experimental results below, the optimization yields very significant savings in practice, despite of its simplicity.
- Sloth-3 Our second optimization, implemented in Sloth-3, replaces the use of nuXmv and IC3 in Sloth-2 by our own, rather simple model checker working directly on the generated AFA. In particular, our model checker is used whenever no split operation is needed after the preprocessing proposed in our first optimization. It works explicitly with sets of conjunctive state formulae representing the configurations reached. The initial formula and transition formulae are first converted to DNF using the Tseytin procedure. Then a SAT solver—in particular, sat4j [Berre and Parrain 2010]—is used to generate new reachable configurations and to check the final condition. Our experimental results show that using this simple model checking approach can win over the advanced IC3 algorithm on formulae without splitting.
- Sloth-4 Our further optimization, Sloth-4, optimizes Sloth-3 by employing the intensionally minimal successor computation of Section 8.2 within the IC3-based model checking of nuXmv.
- Sloth-5 Finally, Sloth-5 modifies Sloth-4 by replacing the use of nuXmv with the property directed reachability (i.e., IC3) implementation in the ABC tool [Brayton and Mishchenko 2010].

We present data on two benchmark groups (each consisting of two benchmark sets) that demonstrate two points. First, the main strength of our tool is shown on solving complex combinations of transducer and concatenation constraints (generated from program code similar to that of Example 1.1) that are beyond capabilities of any other solver. Second, we show that our tool is competitive also on simpler examples that can be handled by other tools (smaller constraints with less intertwined and general combinations of rational and concatenation constraints). All the benchmarks fall within the decidable straight-line fragment (possibly extended with the restricted

length constraints). All experiments were executed on a computer with Intel Xeon E5-2630v2 CPU @ 2.60 GHz and 32 GiB RAM.

Complex combinations of concatenation and rational constraints. The first set of our benchmarks consisted of 10 formulae (5 sat and 5 unsat) derived manually from the PHP programs available from the web page of the Stranger tool [Yu et al. 2010]. The property checked was absence of the vulnerability pattern `.*<script.*` in the output of the programs. The formulae contain 7–42 variables (average 21) and 7–38 atomic constraints (average 18). Apart from the Boolean connectives $\wedge$ and $\vee$, they use regular constraints, concatenation, the `str.replaceall` operation, and several special-purpose transducers encoding various PHP functions used in the programs (e.g., `addslashes`, `trim`, etc.).

Results of running the different versions of Sloth on the formulae are shown in Table 1. Apart from the Sloth version used, the different columns show numbers of solved sat/unsat formulae (together with the time used), numbers of out-of-memory runs ("mo"), as well as numbers of sat/unsat instances for which the particular Sloth version provided the best result ("win +/-"). We can see that Sloth was able to solve 9 out of the 10 formulae, and that each of its versions—apart from Sloth-4—provided the best result in at least some case.

Table 1. PHP benchmarks from the web of Stranger.

Program	#sat (sec)	#unsat (sec)	#mo	#win +/-
Sloth-1	4 (178)	5 (6,989)	1	1/0
Sloth-2	4 (83)	5 (5,478)	1	0/2
Sloth-3	4 (72)	5 (3,673)	1	1/2
Sloth-4	4 (93)	4 (6,168)	2	0/0
Sloth-5	4 (324)	4 (4,409)	2	2/1

Our second benchmark consists of 8 challenging formulae taken from the paper [Kern 2014] providing an overview of XSS vulnerabilities in JavaScript programs (including the motivating example from the introduction).

The formulae contain 9–12 variables (average 9.75) and 9–13 atomic constraints (average 10.5). Apart from conjunctions, they use regular constraints, concatenation, `str.replaceall`, and again several special-purpose transducers encoding various JavaScript functions (e.g., `htmlescape`, `escapeString`, etc.). The results of our experiments are shown in Table 2. The meaning of the columns is the same as in Table 1 except that we drop the out-of-memory column since Sloth could handle all the formulae—which we consider to be an excellent result.

Table 2. Benchmarks from [Kern 2014].

Solver	#sat (sec)	#unsat (sec)	#win +/-
Sloth-1	4 (458)	4 (583)	0/2
Sloth-2	4 (483)	4 (585)	0/1
Sloth-3	4 (508)	4 (907)	2/1
Sloth-4	4 (445)	4 (1,024)	1/0
Sloth-5	4 (568)	4 (824)	1/0

These results are the highlight of our experiments, taking into account that we are not aware of any other tool capable of handling the logic fragment used in the formulae.[9]

A Comparison with other tools on simpler benchmarks. Our next benchmark consisted of 3,392 formulae provided to us by the authors of the Stranger tool. These formulae were derived by Stranger from real web applications analyzed for security; to enable other tools to handle the benchmarks, in the benchmarks the `str.replaceall` operation was approximated by `str.replace`.

[9]We tried to replace the special-purpose transducers by a sequence of `str.replaceall` operations in order to match the syntactic fragment of the S3P solver [Trinh et al. 2016]. However, neither Sloth nor S3P could handle the modified formulae. We have not experimented with other semi-decision procedures, such as those implemented within Stranger or SLOG [Wang et al. 2016], since they are indeed a different kind of tool, and, moreover, often are not able to process input in the SMT-LIBv2 format, which would complicate the experiments.

Apart from the ∧ and ∨ connectives, the formulae use regular constraints, concatenation, and the `str.replace` operation. They contain 1–211 string variables (on average 6.5) and 1–182 atomic formulae (on average 5.8). Importantly, the use of concatenation is much less intertwined with `str.replace` than it is with rational constraints in benchmarks from Tables 1 and 2 (only about 120 from the 3,392 examples contain `str.replace`). Results of experiments on this benchmark are shown in Table 3. In the table, we compare the different versions of our Sloth, the S3P solver, and the CVC4 string solver [Liang et al. 2014].[10] The meaning of the columns is the same as in the previous tables, except that we now specify both the number of time-outs (for a time-out of 5 minutes) and out-of-memory runs ("to/mo").

From the results, we can see that CVC4 is winning, but (1) unlike Sloth, it is a semi-decision procedure only, and (2) the formulae of this benchmark are much simpler than in the previous benchmarks (from the point of view of the operations used), and hence the power of Sloth cannot really manifest.

Table 3. Benchmarks from Stranger with `str.replace`.

Solver	#sat (sec)	#unsat (sec)	#to/mo	#win +/-
Sloth-1	1,200 (19,133)	2,079 (3,276)	105/8	30/43
Sloth-2	1,211 (13,120)	2,079 (3,338)	97/5	19/0
Sloth-3	1,290 (6,619)	2,082 (1,012)	14/6	263/592
Sloth-4	1,288 (6,240)	2,082 (1,030)	17/5	230/327
Sloth-5	1,291 (6,460)	2,082 (953)	14/5	768/1,120
CVC4	1,297 (857)	2,082 (265)	13/0	–
S3P	1,291 (171)	2,078 (56)	13/0	–

Despite that, our solver succeeds in almost the same number of examples as CVC4, and it is reasonably efficient. Moreover, a closer analysis of the results reveals that our solver won in 16 sat and 3 unsat instances. Compared with S3P, Sloth won in 22 sat and 4 unsat instances (plus S3P provided 8 unknown and 1 wrong answer and also crashed once). This shows that Sloth can compete with semi-decision procedures at least in some cases even on a still quite simple fragment of the logic it supports.

Our final set of benchmarks is obtained from the third one by filtering out the 120 examples containing `str.replace` and replacing the `str.replace` operations by `str.replaceall`, which reflects the real semantics of the original programs. This makes the benchmarks more challenging, although they are still simple compared to those of Tables 1 and 2. The results are shown in Table 4. The meaning of the columns is the same as in the previous tables. We compare the different versions of Sloth against S3P only since CVC4 does not support `str.replaceall`. On the examples, S3P crashed 6 times and provided 6 times the unknown result and 13 times a wrong result. Overall, although Sloth is still slower, it is more reliable than S3P (roughly 10 % of wrong and 10 % of inconclusive results for S3P versus 0 % of wrong and 5 % of inconclusive results for Sloth).

Table 4. Benchmarks from Stranger with `str.replaceall`.

Program	#sat (sec)	#unsat (sec)	#to/mo	#win +/-
Sloth-1	101 (1,404)	13 (18)	6/0	9/1
Sloth-2	104 (1,178)	13 (18)	3/0	8/5
Sloth-3	103 (772)	13 (19)	4/0	10/1
Sloth-4	101 (316)	13 (23)	6/0	24/2
Sloth-5	102 (520)	13 (20)	5/0	52/4
S3P	86 (11)	6 (26)	0/5	–

As a final remark, we note that, apart from experimenting with the Sloth-1–5 versions, we also tried a version obtained from Sloth-3 by replacing the intensionally minimal successor computation of Section 8.2 by the deterministic successor computation of Section 8.3. On the given benchmark, this version provided 3 times the best result. This underlines the fact that all of the described optimizations can be useful in some cases.

[10] The S3P solver and CVC4 solvers are taken as two representatives of semi-decision procedures for the given fragment with input from SMT-LIBv2.

10 CONCLUSIONS

We have presented the first practical algorithm for solving string constraints with concatenation, general transduction, and regular constraints; the algorithm is at the same time a decision procedure for the acyclic fragment AC of intersection of rational relations of [Barceló et al. 2013] and the straight-line fragment SL of [Lin and Barceló 2016]. The algorithm uses novel ideas including alternating finite automata as symbolic representations and the use of fast model checkers like IC3 [Bradley 2012] for solving emptiness of alternating automata. In initial experiments, our solver has shown to compare favourably with existing string solvers, both in terms of expressiveness and performance. More importantly, our solver can solve benchmarking examples that cannot be handled by existing solvers.

There are several avenues planned for future work, including more general integration of length constraints and support for practically relevant operations like splitting at delimiters and `indexOf`. Extending our approach to incorporate a more general class of length constraints (e.g. Presburger-expressible constraints) seems to be rather challenging since this possibly would require us to extend our notion of alternating finite automata with *monotonic counters* (see [Lin and Barceló 2016]), which (among others) introduces new problems on how to solve language emptiness.

ACKNOWLEDGMENTS

Holík and Janků were supported by the Czech Science Foundation (project 16-24707Y). Holík, Janků, and Vojnar were supported by the internal BUT grant agency (project FIT-S-17-4014) and the IT4IXS: IT4Innovations Excellence in Science (project LQ1602). Lin was supported by European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant Agreement no 759969). Rümmer was supported by the Swedish Research Council under grant 2014-5484.

REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *CAV*. 150–166.

Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *S&P*. 387–401.

Pablo Barceló, Diego Figueira, and Leonid Libkin. 2013. Graph Logics with Rational Relations. *Logical Methods in Computer Science* 9, 3 (2013). DOI: http://dx.doi.org/10.2168/LMCS-9(3:1)2013

Pablo Barceló, Leonid Libkin, A. W. Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4 (2012), 31.

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proc. of SMT'10*.

Clark W. Barrett, Cesare Tinelli, Morgan Deters, Tianyi Liang, Andrew Reynolds, and Nestan Tsiskaridze. 2016. Efficient solving of string constraints for security analysis. In *Proceedings of the Symposium and Bootcamp on the Science of Security, Pittsburgh, PA, USA, April 19-21, 2016*. 4–6. DOI: http://dx.doi.org/10.1145/2898375.2898393

Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *JSAT* 7, 2-3 (2010), 59–6. http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_4_LeBerre.pdf

Jean Berstel. 1979. *Transductions and Context-Free Languages*. Teubner-Verlag.

Armin Biere, Keijo Heljanko, and Siert Wieringa. 2017. AIGER 1.9 and Beyond (Draft). http://fmv.jku.at/hwmcc11/beyond1.pdf (cited in 2017). (2017).

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *TACAS*. 307–321.

Aaron R. Bradley. 2012. Understanding IC3. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. 1–14. DOI: http://dx.doi.org/10.1007/978-3-642-31612-8_1

Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 24–40. DOI: http://dx.doi.org/10.1007/978-3-642-14295-6_5

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2 (2008), 10:1–10:38. DOI: http://dx.doi.org/10.1145/1455518.1455522

Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 1066–1071. DOI: http://dx.doi.org/10.1145/1985793.1985995

Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *CAV'14 (Lecture Notes in Computer Science)*, Vol. 8559. Springer, 334–342.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. The MIT Press, Cambridge, Massachusetts.

Google co. 2015. Google Closure Library (referred in Nov 2015). https://developers.google.com/closure/library/. (2015).

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Arlen Cox and Jason Leasure. 2017. Model Checking Regular Language Constraints. *CoRR* abs/1708.09073 (2017). arXiv:1708.09073 http://arxiv.org/abs/1708.09073

Loris D'Antoni, Zachary Kincaid, and Fang Wang. 2016. A Symbolic Decision Procedure for Symbolic Alternating Finite Automata. *CoRR* abs/1610.01722 (2016). http://arxiv.org/abs/1610.01722

Loris D'Antoni and Margus Veanes. 2013. Static Analysis of String Encoders and Decoders. In *VMCAI*. 209–228.

Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.

Volker Diekert. 2002. Makanin's Algorithm. In *Algebraic Combinatorics on Words*, M. Lothaire (Ed.). Encyclopedia of Mathematics and its Applications, Vol. 90. Cambridge University Press, Chapter 12, 387–442.

Laurent Doyen and Jean-François Raskin. 2010. Antichain Algorithms for Finite Automata. In *TACAS'10 (Lecture Notes in Computer Science)*, Vol. 6015. Springer, 2–22. DOI: http://dx.doi.org/10.1007/978-3-642-12002-2_2

Alain Finkel. 1987. A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems. In *Automata, Languages and Programming, 14th International Colloquium, ICALP87, Karlsruhe, Germany, July 13-17, 1987, Proceedings (Lecture Notes in Computer Science)*, Thomas Ottmann (Ed.), Vol. 267. Springer, 499–508. DOI: http://dx.doi.org/10.1007/3-540-18088-5_43

Xiang Fu and Chung-Chih Li. 2010. Modeling Regular Replacement for String Constraint Solving. In *NFM*. 67–76.

Xiang Fu, Michael C. Powell, Michael Bantegui, and Chung-Chih Li. 2013. Simple linear string constraints. *Formal Asp. Comput.* 25, 6 (2013), 847–891.

Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. 2013. Word equations with length constraints: what's decidable? In *Hardware and Software: Verification and Testing*. Springer, 209–226.

Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. 2013. Unbounded Model-Checking with Interpolation for Regular Language Constraints. In *TACAS'2013 (Lecture Notes in Computer Science)*, Vol. 7795. Springer, 277–291.

Seymour Ginsburg and Edwin H. Spanier. 1966. Semigroups, Presburger formulas, and languages. *Pacific J. Math.* 16, 2 (1966), 285–296. http://projecteuclid.org/euclid.pjm/1102994974

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 213–223. DOI: http://dx.doi.org/10.1145/1065010.1065036

Claudio Gutiérrez. 1998. Solving Equations in Strings: On Makanin's Algorithm. In *LATIN*. 358–373.

Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z. Yang. 2013. mXSS attacks: attacking well-secured web-applications by using innerHTML mutations. In *CCS*. 777–788.

Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. 2011. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*. http://static.usenix.org/events/sec11/tech/full_papers/Hooimeijer.pdf

Pieter Hooimeijer and Westley Weimer. 2012. StrSolve: solving string constraints lazily. *Autom. Softw. Eng.* 19, 4 (2012), 531–559.

Artur Jez. 2016. Recompression: A Simple and Powerful Technique for Word Equations. *J. ACM* 63, 1 (2016), 4:1–4:51. DOI: http://dx.doi.org/10.1145/2743014

Scott Kausler and Elena Sherman. 2014. Evaluation of String Constraint Solvers in the Context of Symbolic Execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 259–270. DOI: http://dx.doi.org/10.1145/2642937.2643003

Christoph Kern. 2014. Securing the Tangled Web. *Commun. ACM* 57, 9 (Sept. 2014), 38–47.

Adam Kiezun and others. 2012. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4 (2012), 25.

Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. 2002. MONA Implementation Secrets. *International Journal of Foundations of Computer Science* 13, 4 (2002), 571–586.

Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. 2013. Incremental, Inductive Coverability. In *CAV'13 (Lecture Notes in Computer Science)*, Vol. 8044. Springer, 158–173.

Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures.* Springer.

Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *CAV*. 646–662.

Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2016. An efficient SMT solver for string constraints. *Formal Methods in System Design* 48, 3 (2016), 206–234. DOI: http://dx.doi.org/10.1007/s10703-016-0247-6

Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2015. A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings. In *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings.* 135–150. DOI: http://dx.doi.org/10.1007/978-3-319-24246-0_9

Anthony Widjaja Lin and Pablo Barceló. 2016. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *POPL*. 123–136. DOI: http://dx.doi.org/10.1145/2837614.2837641

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In *SPIN*.

Gennady S Makanin. 1977. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics* 32, 2 (1977), 129–198.

John McCarthy. 1980. Circumscription - A Form of Non-Monotonic Reasoning. *Artif. Intell.* 13, 1-2 (1980), 27–39. DOI: http://dx.doi.org/10.1016/0004-3702(80)90011-9

Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings.* 1–13. DOI: http://dx.doi.org/10.1007/978-3-540-45069-6_1

Christophe Morvan. 2000. On Rational Graphs. In *FoSSaCS*. 252–266.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. Abstract DPLL and Abstract DPLL Modulo Theories. In *LPAR'04 (LNCS)*, Vol. 3452. Springer, 36–50.

OWASP. 2013. https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf. (2013).

Wojciech Plandowski. 2004. Satisfiability of word equations with constants is in PSPACE. *J. ACM* 51, 3 (2004), 483–496.

Wojciech Plandowski. 2006. An efficient algorithm for solving word equations. In *STOC*. 467–476.

Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. 2012. Symbolic execution of programs with strings. In *SAICSIT*. 139–148.

Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LNCS)*, Vol. 5330. Springer, 274–289.

Jacques Sakarovitch. 2009. *Elements of automata theory.* Cambridge University Press.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *S&P*. 513–528.

Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013.* 488–498. DOI: http://dx.doi.org/10.1145/2491411.2491447

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *FMCAD (LNCS)*, Vol. 1954. Springer, 108–125.

Deian Tabakov and Moshe Y. Vardi. 2005. Experimental Evaluation of Classical Automata Constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings (Lecture Notes in Computer Science)*, Geoff Sutcliffe and Andrei Voronkov (Eds.), Vol. 3835. Springer, 396–411. DOI: http://dx.doi.org/10.1007/11591191_28

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *CCS*. 1232–1243.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2016. Progressive Reasoning over Recursively-Defined Strings. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I.* 218–240. DOI: http://dx.doi.org/10.1007/978-3-319-41528-4_12

Moshe Y. Vardi. 1995. An Automata-Theoretic Approach to Linear Temporal Logic. In *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, August 27 - September 3, 1995, Proceedings).* 238–266. DOI: http://dx.doi.org/10.1007/3-540-60915-6_6

Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjørner. 2012. Symbolic finite state transducers: algorithms and applications. In *POPL*. 137–150.

Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. 2016. String Analysis via Automata Manipulation with Logic Circuit Representation. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 9779. Springer, 241–260. DOI:http://dx.doi.org/10.1007/978-3-319-41528-4

Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. 2008. Dynamic test input generation for web applications. In *ISSTA*. 249–260.

Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Eui Chul Richard Shin, and Dawn Song. 2011. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *ESORICS*. 150–171.

Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. Stranger: An Automata-Based String Analysis Tool for PHP. In *TACAS*. 154–157. Benchmark can be found at http://www.cs.ucsb.edu/~vlab/stranger/.

Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design* 44, 1 (2014), 44–70.

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. 2009. Symbolic String Verification: Combining String Analysis and Size Analysis. In *TACAS*. 322–336.

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. 2011. Relational String Verification Using Multi-Track Automata. *Int. J. Found. Comput. Sci.* 22, 8 (2011), 1909–1924.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a Z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE*. 114–124.

Regex Matching with Counting-Set Automata

LENKA TUROŇOVÁ*, Brno University of Technology, Czech Republic
LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic
ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic
OLLI SAARIKIVI, Microsoft, USA
MARGUS VEANES, Microsoft, USA
TOMÁŠ VOJNAR, Brno University of Technology, Czech Republic

We propose a solution to the problem of efficient matching regular expressions (regexes) with bounded repetition, such as `(ab){1,100}`, using deterministic automata. For this, we introduce novel *counting-set automata (CsAs)*, automata with registers that can hold sets of bounded integers and can be manipulated by a limited portfolio of constant-time operations. We present an algorithm that compiles a large sub-class of regexes to deterministic CsAs. This includes (1) a novel Antimirov-style translation of regexes with counting to *counting automata (CAs)*, nondeterministic automata with bounded counters, and (2) our main technical contribution, a determinization of CAs that outputs CsAs. The main advantage of this workflow is that the size of the produced CsAs does not depend on the repetition bounds used in the regex (while the size of the DFA is exponential to them). Our experimental results confirm that deterministic CsAs produced from practical regexes with repetition are indeed vastly smaller than the corresponding DFAs. More importantly, our prototype matcher based on CsA simulation handles practical regexes with repetition regardless of sizes of counter bounds. It easily copes with regexes with repetition where state-of-the-art matchers struggle.

CCS Concepts: • **Theory of computation** → **Regular languages**; *Quantitative automata*; • **Security and privacy** → **Denial-of-service attacks**; • **Applied computing** → *Document searching*.

Additional Key Words and Phrases: regular expression matching, bounded repetition, ReDos, determinization, Antimirov's derivatives, counting automata, counting-set automata

ACM Reference Format:
Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex Matching with Counting-Set Automata . *Proc. ACM Program. Lang.* 4, OOPSLA, Article 218 (November 2020), 30 pages. https://doi.org/10.1145/3428286

1 INTRODUCTION

Matching *regexes* (regular expressions) is a ubiquitous component of software, used, e.g., for searching, data validation, parsing, finding and replacing, data scraping, or syntax highlighting. It

*The main part of the work was done when the first author was a summer intern at Microsoft Research in Redmond in 2019.

Authors' addresses: Lenka Turoňová, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, ituronova@fit.vutbr.cz; Lukáš Holík, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, holik@fit.vutbr.cz; Ondřej Lengál, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, lengal@fit.vutbr.cz; Olli Saarikivi, MSR, Microsoft, One Microsoft Way, Redmond, 98052, USA, Olli.Saarikivi@microsoft.com; Margus Veanes, MSR, Microsoft, One Microsoft Way, Redmond, 98052, USA, margus@microsoft.com; Tomáš Vojnar, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, vojnar@fit.vutbr.cz.

2475-1421/2020/11-ART218
https://doi.org/10.1145/3428286

is commonly used and natively supported in most programming languages [contributors 2019]. For instance, about 30–40 % of Java, JavaScript, and Python software use regex matching (as reported in multiple studies, see, e.g., [Davis 2019]).

The efficiency of regex matching engines has a significant impact on the overall usability of software applications. Unpredictability of a matcher's performance may lead to catastrophic consequences, witnessed by events such as the recent catastrophic outage of Cloudflare services [Graham-Cumming 2019], caused by a single poorly written regex, and it is a doorway for the so-called ReDoS attack, a denial of service attack based on overwhelming a regex matching engine by providing a specially crafted regex or text. For instance, in 2016, ReDoS caused an outage of StackOverflow [Exchange 2016] or rendered vulnerable websites that used the popular Express.js framework [Baldwin 2016]. Works such as [Davis 2019; Davis et al. 2018] give arguments that ReDoS is not just a niche problem but rather a common and serious threat.

Failures of matching are mostly caused by the so-called "catastrophic backtracking", a situation when variants of Spencer's simulation of a *nondeterministic finite automaton* (NFA) by *backtracking* [Spencer 1994] exhibit a behaviour super-linear to the length of the text. Matching algorithms based on backtracking are probably the most often implemented ones, their performance is, however, at worst *exponential* to the text length. An alternative with a much lower worst-case complexity (wrt the length of the text) is to use *deterministic finite automata* (DFAs). In the ideal case, the DFA is precomputed; matching can then be linear to the text length, with each input symbol processed in constant time. This is the so-called *static DFA simulation* [Sipser 2006]. The major drawback of static DFA simulation is that the DFA construction may explode, rendering the method unusable in practice.

Variants of Thompson's algorithm [Thompson 1968] (sometimes called NFA simulation or NFA-to-DFA simulation) avoid the explosion by working directly with the NFA. They essentially run the determinization by subset construction *on the fly*, always remembering only the current DFA state. On reading a character, a successor DFA state is computed and used to replace the current state. The disadvantage of Thompson's algorithm is that, for a highly nondeterministic NFA, the DFA states—sets of the states of the NFA—may get large and computing a DFA-state successor over a symbol becomes expensive, linear to the size of the NFA (compared to the constant time of static DFA simulation).

Modern matchers therefore use caching of already visited parts of the DFA. Making a step within the cached part is then as fast as with the explicitly determinized automaton. Extremely efficient implementations of Thompson's algorithm with caching are used in `RE2` [Google [n.d.]] and GNU `grep` [Haertel et al. [n.d.]]. Their close cousin, an on-the-fly Brzozowski's derivative construction, is implemented in the tool `SRM` [Saarikivi et al. 2019]. Highly nondeterministic regexes[1] that lead to exploding determinization are, however, problematic for all variants, explicit determinization as well as NFA simulation, with or without caching.

In this paper, we focus on eliminating a frequent cause of a DFA explosion—a use of the *counting operator*, also known as the operator of *bounded repetition*. It succinctly expresses repeated patterns such as `(ab){1,100}`, representing 1 to 100 consecutive repetitions of `ab`. Such expressions are very common (cf. [Björklund et al. 2015]), e.g., in the RegExLib library [RegExLib.com [n.d.]], which collects expressions for recognizing URIs, markup code, pieces of Java code, or SQL queries; in the Snort rules [M. Roesch et al. [n.d.]] used for detecting attacks in network traffic; in real-life XML schemas, with the counter bounds being as large as 10 million [Björklund et al. 2015]; or in detecting information leakage from traffic logs [Holík et al. 2019].

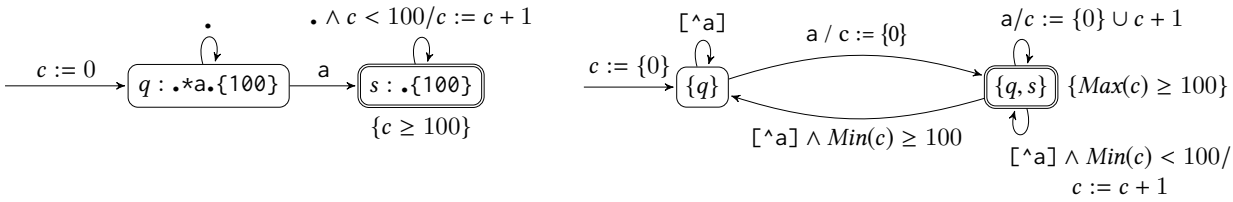
[1]Loosely speaking, a "highly nondeterministic regex" is one for which the determinization of the NFA created by some of the usual algorithms explodes. Determinism of regexes closely corresponds to the notion of *1-unambiguity of the regex* [Brüggemann-Klein and Wood 1998; Hovland 2009]: when matching a text from left to right against the regex, it is always clear which letter of the regex matches the text character.

To illustrate the principal difficulty with matching bounded repetitions, especially when combined with a high degree of nondeterminism, consider the regex `.*a.{`k`}` where $k \in \mathbb{N}$ (the regex denotes strings where the symbol `a` appears k positions from the end of the word). Already the NFA will have at least k states, which is exponential to the regex size because k is written in decimal. Due to the inherent nondeterminism of this regex, determinization then adds a second level of the exponential explosion. Indeed, the minimal DFA accepting the language has 2^{k+1} states because it must remember all the positions where the symbol `a` was seen during the last $k+1$ steps. This requires a finite memory of $k + 1$ bits and thus 2^{k+1} reachable DFA states. Determinizing the NFA explicitly is thus out of question for even moderate values of k. The pure Thompson's NFA simulation is feasible but very slow, as reading each character may in the worst case require processing the entire NFA. Moreover, caching of the DFA state space, used in industrial matchers like `RE2` [Google [n.d.]] or GNU `grep` [Haertel et al. [n.d.]], may also be ineffective due to the size of the state space and low cache utilization. At the same time, combinations of nondeterminism and counting are fairly common. A high degree of nondeterminism is, for instance, usual when searching for a pattern "anywhere on the line" (corresponding to prefixing the pattern with `.*`), which is the standard behaviour for GNU `grep` and similar programs when start/end of line anchors are not used.

To facilitate efficient matching of such nondeterministic counting, we propose a translation from regexes with repetition to deterministic machines that are succinct and can perform matching with nearly constant character complexity. The novel succinct and fast deterministic machine, called the *counting-set automaton (CsA)*, is the key to the result. It is a deterministic finite automaton with a special type of registers that can hold values called *counting sets*—a set of bounded integer values—and support a limited selection of simple set operations. Crucially for the efficiency of our approach, we show that, using a suitable data structure, all the set operations can be implemented to run in *constant time* regardless of the size of the set.

Our compilation from regexes to CsAs proceeds in two steps. First, we compile the regexes into nondeterministic *counting automata (CAs)*, automata with counters whose values are *a priori* bounded. Variants of CAs have been used in several other works under different names, e.g., [Björklund et al. 2015; Gelade et al. 2012; Holík et al. 2019; Hovland 2009; Kilpeläinen and Tuhkanen 2007; Smith et al. 2008b; Sperberg-McQueen [n.d.]]. The compilation from regexes is cheap and produces automata whose size is independent of the counter bounds and linear in the size of the regex. We present a novel translation of regexes to CAs that generalizes the Antimirov's derivative construction [Antimirov 1996]. Our translation has several advantages over the existing alternatives, such as absence of ϵ-transitions in the output CA and succinctness. The result of translating the regex `.*a.{`k`}` into a CA is illustrated in Fig. 1a.

The main step forward we make in this paper is a solution of efficient matching for a large class of highly nondeterministic regexes with counting that are quite common in practice. The main technical problem we have solved is a succinct transformation of a (nondeterministic) CA into a *deterministic CsA*. Our algorithm produces a CsA in *time independent of the counter bounds*. We note that this has been a known open problem (emphasized, e.g., in [Sperberg-McQueen [n.d.]]). Works on matching of bounded repetition such as [Björklund et al. 2015; Gelade et al. 2012; Holík et al. 2019; Hovland 2009; Kilpeläinen and Tuhkanen 2003; Kilpeläinen and Tuhkanen 2007; Smith et al. 2008a] mostly focus on deterministic regexes and do not propose practical solutions for the nondeterministic case. We have carried out an extensive experimental evaluation of our algorithm on a large sample of regexes used for pattern matching in various applications. The experiments show that our algorithm, although also limited to a sub-class of regexes, handles over 90 % of regexes with counting we collected. The obtained data confirm that our CsAs are indeed far smaller and can be constructed faster than corresponding DFAs. Most importantly, we demonstrate the practical



(a) The CA for the regex .*a.{k}

(b) The CsA from determinization of (a)

Fig. 1. The nondeterministic CA and the deterministic CsA for .*a.{100}. The transitions are labeled by their *guard*, which gives the character class (in the standard POSIX regex notation, where, e.g., . stands for "any character") and possibly restricts counter values, delimited by "/" from the counter *update*. If a counter does not have the update specified, then the transition does not change its value. In (b), the notation $c + 1$ stands for the set of values obtained by incrementing each value in c and then *removing* values larger than the upper bound 100 of the counter. The edges denoting initial states are labelled with *initial values* of the counters. Final states are labelled with an *acceptance condition*, e.g. $\{c \geq 100\}$ in (a).
The formal counter operations OP_c presented later in Section 4 are in (a) shown as follows: the guard of OP_c is shown in conjunction with the character guard α on the left of the "/", the update of OP_c is shown on the right of "/" in the form of an assignment to c, where INCR_c appears as the right value $c + 1$, EXIT as 0, EXIT1 as 1, and NOOP is omitted.

relevance of our algorithm for pattern matching. We have implemented a matching algorithm based on CsA simulation[2] and compared it with several state-of-the-art matchers, namely, grep [Haertel et al. [n.d.]], RE2 [Google [n.d.]], SRM [Saarikivi et al. 2019], and .NET [Microsoft 2020]. Our results show that problematic highly nondeterministic regexes with counting indeed appear in practice and can also be easily crafted as a ReDoS attack, and that CsAs can efficiently solve most of such problematic cases. For instance, the regex (_a){64999}_a from [Davis et al. 2019] can cause state-of-the-art matchers exceed any reasonable time limit (when searching for the pattern anywhere on the line, with the implicit .* in front). Already with the counter bounds lowered to 1,000, the matchers take from 6 to 34 seconds on 500 KiB of text, but our algorithm needs only 1 second even with the original bound 64,999.

We summarize the technical contributions of this work as follows:

(1) A novel Antimirov style regex-to-CA translation.
(2) A novel notion of the counting-set automaton, a deterministic machine that allows for succinct representation of counting constraints and fast matching.
(3) CA-to-CsA determinization that runs in time independent of counter bounds.
(4) A regex-matching algorithm interconnecting the above, efficient regardless of counter bounds especially on regexes that combine counting with nondeterminism.
(5) Implementation and extensive experimental evaluation of the above.

2 OVERVIEW

We give a brief overview of our conversion of a regex with counting into a deterministic CsA. We use the example regex R = .*a.{100}, discussed already in the introduction and representing strings where the symbol a appears 100 positions from the end, with the corresponding minimum DFA having 2^{101} states. The conversion proceeds in two steps. First, R is translated into a nondeterministic CA (Fig. 1a), denoted as CA(R); second, CA(R) is converted into a deterministic CsA (Fig. 1b). The size of both is independent of the counter bounds (both of the automata will have 2 states only).

[2]We use a pre-computed deterministic CsA. While on-the-fly determinization is also possible, it was not needed in our experimentation since we have not witnessed problems with CsA state space explosion.

Counting-Set Data Structure. Before looking into the conversion from regular expressions to CsAs it is useful to first understand *why* the resulting CsA can be used efficiently for matching in the first place. The main enabler behind this is the use of our *counting-set* data structure, say c, representing sets $S_c \subseteq \{0, \ldots, \boldsymbol{max}_c\}$ where the upper bound $\boldsymbol{max}_c$ is a fixed positive integer. A *runtime value* of c is a tuple (o, ℓ) where $o \in \mathbb{N}$ is called an *offset* and ℓ is a queue of strictly increasing natural numbers such that $S_c = \{o - n \mid n \in \ell\}$.

The data structure supports *constant-time* implementations of the following operations, assuming constant-time access to the first and the last element of the queue (the queue may be implemented as a doubly-linked list).

- The minimum and the maximum of S_c are obtained as $o - \mathit{last}(\ell)$ and $o - \mathit{first}(\ell)$, respectively.
- Insert 0: if $o - \mathit{last}(\ell) > 0$, then append o at the end of ℓ (similarly for inserting 1).
- Increment all, up to $\boldsymbol{max}_c$: $o := o + 1$; if $o - \mathit{first}(\ell) > \boldsymbol{max}_c$, then remove $\mathit{first}(\ell)$.
- Reset to $\{0\}$: $\ell := 0$; $o := 0$ (similarly for reset to $\{1\}$).

The independence of the running time of these operations of $\boldsymbol{max}_c$ enables our major achievement:

the independence of the running time of pattern matching of the counter bounds.

Let us now illustrate how this data structure works during matching. We run the CsA in Fig. 1b, assuming the meaning of the operations provided above, over the sample input word $\texttt{aa0}^{(10)}\texttt{aab}^{(87)}\texttt{dfa}$. The configurations of the automaton after processing prefixes of the word are shown in the table: the control state, the counting-set run-time configuration (o, ℓ), and the value S_c it represents. The state $\{q, s\}$ fulfills the *accepting condition* after processing the 6th and the 7th prefix since the maximum of S_c at these points is indeed at least 100.

prefix	state	(o, ℓ)	S_c
ϵ	$\{q\}$	$(0, [0])$	$\{0\}$
a	$\{q, s\}$	$(0, [0])$	$\{0\}$
aa	$\{q, s\}$	$(1, [0, 1])$	$\{1, 0\}$
$\texttt{aa0}^{(10)}$	$\{q, s\}$	$(11, [0, 1])$	$\{11, 10\}$
$\texttt{aa0}^{(10)}\texttt{aa}$	$\{q, s\}$	$(13, [0, 1, 12, 13])$	$\{13, 12, 1, 0\}$
$\texttt{aa0}^{(10)}\texttt{aab}^{(87)}$	$\{q, s\}$	$(100, [0, 1, 12, 13])$	$\{100, 99, 88, 87\}$
$\texttt{aa0}^{(10)}\texttt{aab}^{(87)}\texttt{d}$	$\{q, s\}$	$(101, [1, 12, 13])$	$\{100, 89, 88\}$
$\texttt{aa0}^{(10)}\texttt{aab}^{(87)}\texttt{df}$	$\{q, s\}$	$(102, [12, 13])$	$\{90, 89\}$
$\texttt{aa0}^{(10)}\texttt{aab}^{(87)}\texttt{dfa}$	$\{q, s\}$	$(103, [12, 13, 103])$	$\{91, 90, 0\}$

From a Nondeterministic CA to a Deterministic CsA. The idea of our CA-to-CsA determinization is best explained by comparison with the naive determinization of a CA, which would create a DFA by the explicit textbook-style subset construction. The states of the DFA would then be sets of runtime configurations of the CA where each CA-configuration would consist of a control state and a counter valuation. Counter valuations would hence be "unfolded"—they would become an explicit part of the DFA control states—and the succinctness provided by counters would be lost. For instance, the run of the CA in Fig. 1a on the word $\texttt{aa0}\ldots$ generates "powerstates"

$$\{(q, c = 0)\},\ \{(q, c = 0), (s, c = 0)\},\ \{(q, c = 0), (s, c = 0), (s, c = 1)\},\ \{(q, c = 0), (s, c = 1), (s, c = 2)\}, \ldots$$

which are essentially subsets of $\{q, s\} \times \{0, \ldots, 100\}$. In the worst case, the size of the DFA would be exponential in counter bounds because s can be paired with any subset of $\{0, \ldots, 100\}$ recording possible values of c. In contrast to this, as illustrated above, our CsA represents the counter valuations implicitly: it computes them dynamically on the fly and stores them as *counting sets*—i.e., the valuation of a counter changes from an integer to a *set* of integers. The counter valuations are hence not a part of control states, and their overall number influences neither the size of the CsA nor the time needed to build them.

Fig. 1b shows the CsA obtained from determinization of the CA in Fig. 1a. The runtime configurations of the CsA reached for the word $\texttt{aa0}$ are

$$(\{q\}, c \in \{0\}),\ \ (\{q, s\}, c \in \{0\}),\ \ (\{q, s\}, c \in \{0, 1\}),\ \ (\{q, s\}, c \in \{1, 2\}).$$

They encode the first three states reached by the sample DFA run above. Namely, the control states are kept in the first component and the counter values are in the second component, i.e., the set S_c given by the run-time values of the counting-set c. In this encoding, the value of the counting-set is not relevant for the states where the counter is never active (state q in our example). The counter's value in these states is always implicitly 0. In the example, the value S_c of the counting-set therefore only records the values of c at state s and is thus relevant only in the CsA state $\{q, s\}$. We note that for simplicity, we initialise all counting-sets uniformly with $\{0\}$, even if their value is initially irrelevant, as in the case of the CsA state $\{q\}$ in the example.

We note that some DFA powerstates cannot be encoded as CsA configurations due to the involved Cartesian abstraction: essentially, any state in the powerset is paired with any counter value from the counting set. Hence, our approach does not handle the full class of regexes. Fortunately, as our empirical evidence shows, regexes that fall out from the supported class are rare in practice.

From Regexes to Nondeterministic CAs. To translate a regex into a CA, we propose a generalization of the Antimirov's derivative construction [Antimirov 1996] to symbolic counting. In Antimirov's setting, a derivative of a regex R wrt a character class α is a set of regexes that together capture all tails of words in $\mathcal{L}(R)$ whose head character is from α. In particular, according to [Saarikivi et al. 2019], which generalizes [Antimirov 1996] to *explicit* counting, the derivatives of the regex $R =$ `.*a.{100}` wrt the classes `a` and `[^a]` are $\{R,$ `.{100}`$\}$ and $\{R\}$, respectively. Further, for $1 \leq k \leq 100$, the derivative of `.{`k`}` wrt both `a` and `[^a]` is {`.{`$k-1$`}`}. The derivatives become the states of the resulting NFA, with R being initial and `.{0}` final, and with α-transitions from each regex to all its α-derivatives (for α being either `a` or `[^a]`). The obtained NFA is already large, it has 102 states.

In our new construction, the counting is kept *implicit* using symbolic counters. Instead of modifying the counter bound of the derivative (by, e.g., deriving `.{99}` from `.{100}`), we keep the original bound unchanged and use a counter c to keep track of the difference between the original value and the current value. Our *conditional derivative* operator $\partial_\alpha(\cdot)$ then equips the produced derivatives with *conditional counter updates* to keep the counters up-to-date. For instance, $\partial_{\texttt{a}}($`.{100}`$)$ returns the same regex `.{100}`, but it is paired with conditional counter updates for c, namely, "if $c < 100$, then increment c; and if $c \geq 100$, then exit the counting loop". The CA we obtain this way is shown in Fig. 1a, where the first conditional update translates to the self loop on the state `.{100}` and the second to the acceptance condition. The size of the CA does not depend on the counter bounds.

3 PRELIMINARIES

We cast our definitions in the framework of symbolic automata [D'Antoni and Veanes 2020], a natural succinct representations of finite-state transition relations over large alphabets of labels. Symbolic automata work over alphabets equipped with a so-called effective Boolean algebra, which defines the needed interface for handling large sets of labels on automata transitions.

Before providing the formal definition of an effective Boolean algebra, we start with an intuitive example, which is also going to be the alphabet algebra used throughout the paper, including the experiments. Later on, we will further leverage the general definition to work with algebras over counter and counting-set predicates.

Example 3.1. Regular expressions in practice use *character classes* as basic building blocks. To simplify the discussion, let us restrict our attention to ASCII as the character universe $\mathfrak{D}$. In other words, $\mathfrak{D}$ is the set $\{n \mid 0 \leq n < 2^7\}$ of all 7-bit characters represented using their character codes. Then, for example, the character classes `[0-9]` and `[A-Z]` denote, respectively, the set $[\![$`[0-9]`$]\!] = \{n \mid 48 \leq n \leq 57\}$ of all digit codes, and the set $[\![$`[A-Z]`$]\!] = \{n \mid 65 \leq n \leq 90\}$ of all upper-case letter codes. Character classes made up of individual symbols such as `@` denote singleton sets, e.g., $[\![$`@`$]\!] = \{64\}$. Character classes can also be used to form *unions*, they can be *complemented*, and even *subtracted* from each other, and are in general closed under Boolean

operations. There are therefore many different ways how to represent the same character sets, e.g., $[\![\texttt{[0-9]}]\!] = [\![\texttt{[0-45-9]}]\!] = [\![\texttt{[0-4]}]\!] \cup [\![\texttt{[5-9]}]\!]$. To illustrate the complement, for example, `[^0-9]` denotes the set of all non-digits, as does `[\x00-\x2F\x3A-\x7F]`. The set of all character classes is then an example of the set Ψ of all *predicates* of a Boolean algebra, and checking *satisfiability* of a predicate $\varphi \in \Psi$ means to decide whether φ denotes a *nonempty* set. For example, the predicate `[]` is unsatisfiable because $[\![\texttt{[]}]\!] = \emptyset$ and `.` denotes the *true* predicate because $[\![\texttt{.}]\!] = \mathfrak{D}$. Further, note that a character class can, without loss of generality, be represented as a Boolean combination of *intervals* or even as a union of intervals if normalized. □

3.1 Effective Boolean Algebras

An *effective Boolean algebra* $\mathbb{A}$ has components $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is a *universe* of underlying domain elements. Ψ is a set of unary *predicates* closed under the Boolean connectives $\vee, \wedge : \Psi \times \Psi \to \Psi$ and $\neg : \Psi \to \Psi$; and $\bot, \top \in \Psi$ are the *false* and *true* predicates. Values of the algebra are sets of domain elements, and the *denotation function* $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ satisfies that $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, and for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \Psi$, we write $\textbf{\textit{Sat}}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$, and we say that φ is *satisfiable*. We require that $\textbf{\textit{Sat}}$ as well as $\vee$, $\wedge$, and $\neg$ are *computable* as a part of the definition of an effective Boolean algebra. We write $x \models \varphi$ for $x \in [\![\varphi]\!]$ and we use $\mathbb{A}$ as a subscript of a component when it is not clear from the context, e.g., $[\![\_]\!]_{\mathbb{A}} : \Psi_{\mathbb{A}} \to 2^{\mathfrak{D}_{\mathbb{A}}}$.

3.2 Words and Regexes

The basic building blocks of regexes are *predicates* from an effective Boolean algebra *CharClass* of *character classes*, such as the class of digits, written as `\d`. Let $\mathfrak{D} = \mathfrak{D}_{CharClass}$. A *word* over $\mathfrak{D}$ is a sequence of symbols $a_1 \cdots a_n \in \mathfrak{D}^*$ and a *language* $\mathcal{L}$ over $\mathfrak{D}$ is a subset of $\mathfrak{D}^*$. We use ϵ to denote the *empty word*. The concatenation of words u and v is denoted as $u \cdot v$ (often abbreviated to uv) and is lifted to sets as usual. We call $a \in \mathfrak{D}$ the *head* of the word $a.w$ and $w \in \mathfrak{D}^*$ its *tail*. Furthermore, we write $\mathcal{L}^n$ for the n-th power of $\mathcal{L} \subseteq \mathfrak{D}^*$ with $\mathcal{L}^0 \stackrel{\text{def}}{=} \{\epsilon\}$ and $\mathcal{L}^{n+1} \stackrel{\text{def}}{=} \mathcal{L}^n \cdot \mathcal{L}$.

The syntax of regexes is the following, with $\alpha \in \Psi_{CharClass}$ and $n, m \in \mathbb{N}$, $0 \leq n$, $0 < m$, $n \leq m$:

$$\varepsilon \quad \alpha \quad R_1 \cdot R_2 \quad R_1|R_2 \quad R\{n,m\} \quad R*$$

where $R_1 \cdot R_2$ is called a *concat node* and $R_1|R_2$ is called a *choice node*. The semantics of a regex R is defined as a subset of $\mathfrak{D}^*$ in the following way: $\mathcal{L}(\alpha) \stackrel{\text{def}}{=} [\![\alpha]\!]$, $\mathcal{L}(\varepsilon) \stackrel{\text{def}}{=} \{\epsilon\}$, $\mathcal{L}(R_1R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$, $\mathcal{L}(R_1|R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, $\mathcal{L}(R\{n,m\}) \stackrel{\text{def}}{=} \bigcup_{i=n}^{m} (\mathcal{L}(R))^i$, and $\mathcal{L}(R*) \stackrel{\text{def}}{=} \mathcal{L}(R)^*$. R is *nullable* if $\epsilon \in \mathcal{L}(R)$. We will also need to refer to the number of *character-class leaf nodes* of a regex R, denoted by $\sharp_\Psi(R)$ and defined as follows: $\sharp_\Psi(\varepsilon) = 0$, $\sharp_\Psi(\alpha) = 1$, $\sharp_\Psi(R_1 \cdot R_2) = \sharp_\Psi(R_1|R_2) = \sharp_\Psi(R_1) + \sharp_\Psi(R_2)$, $\sharp_\Psi(R\{n,m\}) = \sharp_\Psi(R*) = \sharp_\Psi(R)$.

3.3 Minterms

Let $Preds(R)$ be the set of all predicates that occur in a regex R, and let $Minterms(R)$ denote the set of *minterms* of $Preds(R)$. Intuitively, $Minterms(R)$ is a set of non-overlapping predicates that can be treated as a concrete finite alphabet. Each minterm is essentially a region in the Venn diagram of the predicates in R: it is a satisfiable conjunction $\bigwedge_{\psi \in Preds(R)} \psi'$ where $\psi' \in \{\psi, \neg\psi\}$. For example, if $R =$ `[0-z]{4}[0-8]{5}`, then $Preds(R) = \{$`[0-8]`, `[0-z]`$\}$ and $Minterms(R) = \{$`[0-8]`, `[9-z]`, `[^0-z]`$\}$. Formally, if $\alpha \in Minterms(R)$, then $\textbf{\textit{Sat}}(\alpha)$ and $\forall \psi \in Preds(R)$: $[\![\alpha]\!] \cap [\![\psi]\!] \neq \emptyset \Rightarrow [\![\alpha]\!] \subseteq [\![\psi]\!]$.

Note that although the number of minterms of a general set X of predicates may be exponential in $|X|$, it is only linear if X consists of intervals of symbols used in regexes, such as `[a-zA-Z]` or `[^a-zA-Z]` (the former denotes two intervals while the latter their complement, which is equivalent to the union of three intervals). Intervals of numbers generate only a linear number of minterms.

3.4 Symbolic Automata

We use *symbolic finite automata (FAs)*, whose alphabet is given by an effective Boolean algebra, as a generalization of classical finite automata. Formally, an FA is a tuple $A = (\mathbb{I}, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is an effective Boolean algebra called the input algebra, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times Q$ is a finite set of transitions. A transition $(q, \alpha, r) \in \Delta$ will be also written as $q \text{-}(\alpha)\!\!\rightarrow r$.

A *run of A from a state p_0* over a word $a_1 \cdots a_n$ is a sequence of transitions $(p_{i-1} \text{-}(\alpha_i)\!\!\rightarrow p_i)_{i=1}^{n}$ with $a_i \in [\![\alpha_i]\!]$; the run is *accepting* if $p_n \in F$. The *language of a state q*, denoted $\mathcal{L}_A(q)$, is the set of words over which A has an accepting run from q. The *language of A*, denoted $\mathcal{L}(A)$, is $\mathcal{L}_A(q_0)$. A classical finite automaton can be understood as an FA where the basic predicates have singleton set semantics, i.e., when for each concrete letter a there is a predicate α_a such that $[\![\alpha_a]\!] = \{a\}$. A is *deterministic* iff for all $p \in Q$ and all transitions $p \text{-}(\alpha)\!\!\rightarrow q$ and $p \text{-}(\alpha')\!\!\rightarrow r$, it holds that if $\alpha \wedge \alpha'$ is satisfiable, then $q = r$.

4 COUNTING AUTOMATA

Counting automata (CAs) are a natural and compact automata counterpart for regexes with counting. They are essentially a limited sub-class of classical counter automata, in which counters are only supposed to count the number of passes through some of its parts (corresponding to a counted sub-expression of a regex) and guards on transitions enforce a specified number of repetitions of that part before the automaton is allowed to move on.

Counter algebra. A *counter algebra* is an effective Boolean algebra $\mathbb{C}$ associated with a finite set C of counters. The counters play the role of bounded loop variables associated with a *lower bound* $\boldsymbol{min}_c \geq 0$ and an *upper bound* $\boldsymbol{max}_c > 0$ such that $\boldsymbol{min}_c \leq \boldsymbol{max}_c$. $\mathfrak{D}_{\mathbb{C}}$ is the set of interpretations $\mathfrak{m} : C \to \mathbb{N}$, called *counter memories* such that $0 \leq \mathfrak{m}(c) \leq \boldsymbol{max}_c$ for all $c \in C$. $\Psi_{\mathbb{C}}$ contains Boolean combinations of *basic* predicates $\textsc{canExit}_c$ and $\textsc{canIncr}_c$, for $c \in C$, whose semantics is given by

$$\mathfrak{m} \models \textsc{canExit}_c \iff \mathfrak{m}(c) \geq \boldsymbol{min}_c, \quad \mathfrak{m} \models \textsc{canIncr}_c \iff \mathfrak{m}(c) < \boldsymbol{max}_c.$$

Counting automata. A *counting automaton (CA)* is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is an effective Boolean algebra called the *input algebra*, C is a finite set of *counters* with an associated counter algebra $\mathbb{C}$, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $F : Q \to \Psi_{\mathbb{C}}$ is the *final-state condition*, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times (C \to \boldsymbol{O}) \times Q$ is the (finite) *transition relation*, where $\boldsymbol{O} = \{\textsc{exit}, \textsc{incr}, \textsc{exit1}, \textsc{noop}\}$ is the set of *counter operations*. The component f of a transition $p \text{-}(\alpha,f)\!\!\rightarrow q \in \Delta$ is its *(counter) operator*. We often view f as a set of *indexed operations* $\textsc{op}_c$ to denote the operation assigned to the counter c, $f(c) = \textsc{op}$.

Semantics of CAs. The semantics of the CA A is defined through its *configuration automaton FA(A)*, an FA whose states are A's *configurations*, i.e., pairs $(q, \mathfrak{m}) \in Q \times \mathfrak{D}_{\mathbb{C}}$ consisting of a state q and a counter memory $\mathfrak{m}$. To define FA(A), we must first define the semantics of counter operators f, which occur on transitions. For this, we associate with each (indexed) operation $\textsc{op}_c$ a counter guard $grd(\textsc{op}_c)$ and a counter update $upd(\textsc{op})$ as shown on the right. Intuitively, the operation NOOP does not modify the counter's value and is always enabled. The operation INCR increments the counter and is enabled if the counter has not yet reached its upper bound. The operation EXIT resets the counter to 0 on exit from the counting loop and is enabled when the counter reaches its lower bound. The operation EXIT1 executes EXIT followed by INCR. The *guard* of a counter operator $f : C \to \boldsymbol{O}$ is then a predicate $\varphi_f \in \Psi_{\mathbb{C}}$ over

$$\begin{array}{ll|ll} grd(\textsc{noop}_c) & \stackrel{\text{def}}{=} \top_{\mathbb{C}} & upd(\textsc{noop}) & \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n \\ grd(\textsc{incr}_c) & \stackrel{\text{def}}{=} \textsc{canIncr}_c & upd(\textsc{incr}) & \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n+1 \\ grd(\textsc{exit}_c) & \stackrel{\text{def}}{=} \textsc{canExit}_c & upd(\textsc{exit}) & \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.0 \\ grd(\textsc{exit1}_c) & \stackrel{\text{def}}{=} \textsc{canExit}_c & upd(\textsc{exit1}) & \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.1 \end{array}$$

counter memories, and its *update* $\mathrm{f} : \mathfrak{D}_{\mathbb{C}} \cup \{\bot\} \to \mathfrak{D}_{\mathbb{C}} \cup \{\bot\}$ is a counter-memory transformer:

$$\varphi_f \stackrel{\text{def}}{=} \bigwedge_{\mathrm{OP}_c \in f} grd(\mathrm{OP}_c) \qquad \mathrm{f}(\mathfrak{m}) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{\lambda} c.\, upd(f(c))(\mathfrak{m}(c)) & \text{if } \mathfrak{m} \models \varphi_f \\ \bot & \text{otherwise} \end{cases}$$

Intuitively, f updates all counters in a counter memory $\mathfrak{m}$ by their corresponding operations if $\mathfrak{m}$ satisfies the guard, otherwise the result is $\bot$.

We now define the *configuration automaton* of A, denoted as $FA(A)$, which defines the language semantics of the CA A. The states of $FA(A)$ are the configurations of A (there are finitely many of them), and the initial state of $FA(A)$ is the *initial configuration* $(q_0, \{c \mapsto 0 \mid c \in C\})$ of A. A state $(p, \mathfrak{m})$ of $FA(A)$ is *final* iff $\mathfrak{m} \models F(p)$. The transition relation $\Delta_{FA(A)}$ of $FA(A)$ is defined as

$$\Delta_{FA(A)} = \{(p, \mathfrak{m}) \text{-}(\alpha)\!\!\to (q, \mathrm{f}(\mathfrak{m})) \mid p \text{-}(\alpha, f)\!\!\to q \in \Delta, \mathfrak{m} \models \varphi_f\}.$$

Deterministic and simple CAs. A is *deterministic* iff the following holds for every state $p \in Q$ and every two transitions $p \text{-}(\alpha_1, f_1)\!\!\to q_1$, $p \text{-}(\alpha_2, f_2)\!\!\to q_2 \in \Delta$: if both $\alpha_1 \wedge \alpha_2$ and $\varphi_{f_1} \wedge \varphi_{f_2}$ are satisfiable, then $f_1 = f_2$ and $q_1 = q_2$. It follows from the definitions that, if A is deterministic, then $FA(A)$ is deterministic too. A is *simple* if for any two transitions $q \text{-}(\alpha, f)\!\!\to r$ and $q' \text{-}(\alpha', f')\!\!\to r'$, either $\alpha = \alpha'$ or $[\![\alpha]\!] \cap [\![\alpha']\!] = \emptyset$. That is, different character guards do not overlap and can be mostly treated as plain symbols. We also require that all guards are satisfiable. CAs constructed from regexes by the algorithm in Section 5 will be simple.

Example 4.1. Fig. 1a shows a CA in an intuitive notation, with the initial state q and final conditions $F(q) = \bot$, $F(s) = \textsc{exit}_c$, where $\boldsymbol{min}_c = \boldsymbol{max}_c = 100$. The same notation is used in Fig. 2. Fig. 3a shows a CA in a notation following the formal development more closely. □

5 FROM A REGEX TO A CA VIA CONDITIONAL PARTIAL DERIVATIVES

We introduce a generalization of the Antimirov's partial derivative construction [Antimirov 1996] to *symbolic* counting, which allows one to replace a verbose NFA by a succinct CA. The difference between the older variant of [Antimirov 1996] with *explicit* counting [Saarikivi et al. 2019] and our new version was already illustrated in Section 2. To recall it briefly using the example of the regex `.{100}`: from 100 partial derivatives $\partial_{\bullet}(\texttt{.\{i\}}) = \texttt{.\{i-1\}}$, $1 \le i \le 100$, and an NFA with 100 states and transitions $\texttt{.\{i\}} \text{-}(\bullet)\!\!\to \texttt{.\{i-1\}}$, the new construction will take us to the single derivative $\partial_{\bullet}(\texttt{.\{100\}}) = \{\texttt{.\{100\}}\}$ associated with a conditional counter update which induce an NFA with a single state and the transition $\texttt{.\{100\}} \text{-}(\alpha, \textsc{incr}_c)\!\!\to \texttt{.\{100\}}$.

We apply the construction on regexes that are normalized using the below rules where $X \rightsquigarrow Y$ denotes that X is rewritten to Y:

- All nested concat nodes are rewritten to the flattened right-associative *list form*, which is always maintained throughout the construction, using the rules: $(X \cdot Y) \cdot Z \rightsquigarrow X \cdot (Y \cdot Z)$, $\varepsilon \cdot Z \rightsquigarrow Z$, and $Z \cdot \varepsilon \rightsquigarrow Z$.
- If S is *nullable*, then $S\{\ell, k\} \rightsquigarrow S\{0, k\}$. Moreover, in the nullable context $S\{0, k\}$, S can be considered as if it was not nullable.

Observe that the normalization does not increase the size of the regex (it may decrease the size).

Let R be a fixed normalized regex. A sub-expression of R that is of the form $X = S\{\ell, k\}$ is called a *counting loop*. We consider each counting loop to represent a *counter* whose name is the counting loop itself and whose *upper bound* is $\boldsymbol{max}_X = k$ and *lower bound* is $\boldsymbol{min}_X = \ell$. For example, $(\texttt{.\{9\}})*$ contains the counter $X = \texttt{.\{9\}}$ with $\boldsymbol{min}_X = \boldsymbol{max}_X = 9$. In the following, let C stand for the set of all counters that occur in R, also denoted by $Counters(R)$.

We use the convention that the juxtaposition XY of normalized regexes X and Y is again a normalized regex of the equivalent concat node $X \cdot Y$: e.g., if $X = a \cdot b$ and $Y = (a \cdot b)*$, then

$XY = a \cdot (b \cdot (a \cdot b)*)$. Observe in particular that $X\varepsilon = X$. In other words, we treat concatenated elements as sequences, and a singleton sequence equals to the element itself.

Our construction will work over the set $\Sigma = \mathit{Minterms}(R)$ of minterms of R and produce simple CA that use minterms of Σ on transitions.

5.1 Parametric Languages

We define the language of a normalized regex starting with a counting loop relative to a counter value. For that, we lift the definition of languages to be parametric in counter memories $\mathfrak{m}$, but regexes other than the above are treated as before with the memory $\mathfrak{m}$ passed through unchanged.

$$\mathrm{L}^{\mathfrak{m}}(\varepsilon) \stackrel{\text{def}}{=} \{\epsilon\} \quad (1)$$

$$\mathrm{L}^{\mathfrak{m}}(\psi Z) \stackrel{\text{def}}{=} [\![\psi]\!] \cdot \mathrm{L}^{\mathfrak{m}}(Z) \quad (2)$$

$$\mathrm{L}^{\mathfrak{m}}((W|Y)Z) \stackrel{\text{def}}{=} \mathrm{L}^{\mathfrak{m}}(WZ) \cup \mathrm{L}^{\mathfrak{m}}(YZ) \quad (3)$$

$$\mathrm{L}^{\mathfrak{m}}(S*Z) \stackrel{\text{def}}{=} \mathrm{L}^{\mathfrak{m}}(S)^* \cdot \mathrm{L}^{\mathfrak{m}}(Z) \quad (4)$$

$$\mathrm{L}^{\mathfrak{m}}(S\{\ell,k\}Z) \stackrel{\text{def}}{=} \mathrm{L}^{\mathfrak{m}}(S) \cdot \mathrm{L}^{\textsc{incr}_{S\{\ell,k\}}(\mathfrak{m})}(S\{\ell,k\}Z) \cup \mathrm{L}^{\textsc{exit}_{S\{\ell,k\}}(\mathfrak{m})}(Z) \quad (5)$$

$$\mathrm{L}^{\perp}(X) \stackrel{\text{def}}{=} \emptyset \quad (\text{for all } X) \quad (6)$$

Recall that if f is a counter operator and $\mathfrak{m}$ a counter memory, then $\mathrm{f}(\mathfrak{m})$ denotes the appropriately updated memory where $\mathrm{f}(\mathfrak{m}) = \perp$ when f is not enabled in $\mathfrak{m}$. Below, if there is a single counter $c \in C$ such that $f(c) \neq \textsc{noop}$, we sometimes identify f with $\textsc{op}_c$ and use $\textsc{op}_c(\mathfrak{m})$ to represent the updated memory $\mathrm{f}(\mathfrak{m})$. Specifically, $\textsc{incr}_X$ (if enabled) increments the counter value of X by 1, and $\textsc{exit}_X$ (if enabled) resets the counter value of X to 0. Let $\mathfrak{m}$ be a counter memory. Then Cases (1)–(6) define the *parametric languages* of regexes. The intuition behind Case (4) is that all counters that may be present in S are inactive on the level of $S*$. Note also that Case (5) is well-defined since, for $X = S\{\ell,k\}$ and $\mathfrak{m}' = \textsc{incr}_X(\mathfrak{m})$, $k - \mathfrak{m}'(X) < k - \mathfrak{m}(X)$ if $\mathfrak{m}(X) < k$, and $\mathfrak{m}' = \perp$ if $\mathfrak{m}(X) = k$.

Let $0 \stackrel{\text{def}}{=} \boldsymbol{\lambda} c.0$ denote the initial memory that maps all counters to 0. The below theorem, proven in [Turoňová et al. 2020], relates $\mathrm{L}^{\mathfrak{m}}(R)$ with the non-parametric definition of regular languages.

Theorem 5.1. *Let R be a normalized regex. Then $\mathrm{L}^{0}(R) = \mathcal{L}(R)$.*

5.2 Conditional Derivation

We will now introduce our conditional derivative construction formally.

A *partial conditional derivative* is a pair $\langle f, X\rangle$ where f is a counter operator and X a normalized regex. Given a counter memory $\mathfrak{m}$, we let $\langle f, X\rangle$ define the language $\mathrm{L}^{\mathfrak{m}}(\langle f, X\rangle) \stackrel{\text{def}}{=} \mathrm{L}^{\mathrm{f}(\mathfrak{m})}(X)$. In other words, f is first applied to the counter memory $\mathfrak{m}$ and then the regex is evaluated in the updated memory. If f is not enabled in $\mathfrak{m}$, then the denoted language is empty.

A *conditional derivative* is a finite set of partial conditional derivatives. The language defined by a conditional derivative D in a counter memory $\mathfrak{m}$ is defined as the union of the languages of the partial conditional derivatives in D: $\mathrm{L}^{\mathfrak{m}}(D) \stackrel{\text{def}}{=} \bigcup_{d \in D} \mathrm{L}^{\mathfrak{m}}(d)$.

To define how conditional derivatives of a given regex looks like, we need a notion of a *sequential composition* of conditional derivatives $D \otimes E \stackrel{\text{def}}{=} \{\langle f; g, X{\cdot}Y\rangle \mid \langle f, X\rangle \in D, \langle g, Y\rangle \in E, f; g \neq \perp\}$ where $f; g \neq \perp$ is the composed counter operator such that $\mathrm{f}; \mathrm{g}(\mathfrak{m}) = \mathrm{g}(\mathrm{f}(\mathfrak{m}))$. The case when $f; g = \perp$ is discussed later on.

Conditional derivatives of a normalized regex are defined as shown on the right assuming that concatenations $X{\cdot}Y$ are normalized to the list form explained above, $\alpha \in \Sigma$, $\textbf{ID}$ denotes the identity function $\boldsymbol{\lambda} x.x$, and $X = S\{\ell,k\}$ is a counting loop. Observe that, in $\partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\}$, the operation $\textsc{incr}_X$ gets composed with $\textsc{noop}_X$, yielding $\textsc{incr}_X$ again, because $S\{\ell,k\}$ cannot occur in S. It is possible that in $\{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_\alpha(Z)$, X is in scope of Z (e.g., Z starts with X). The composition can then contain the operation $\textsc{exit}_X; \textsc{incr}_X$ that corresponds to $\textsc{exit}1_X$ because $\textsc{incr}_X$ is trivially

enabled when the counter value of X is 0. The only other possible composition of individual operations that can appear in this case is $\textsc{exit}_X; \textsc{exit}_X$. If $\boldsymbol{min}_X = 0$, $\textsc{exit}_X; \textsc{exit}_X = \textsc{exit}_X$, which is well-defined because $\textsc{exit}_X$ is always enabled for $\boldsymbol{min}_X = 0$. If $\boldsymbol{min}_X > 0$, then $\textsc{exit}_X; \textsc{exit}_X$ is undefined, and $\textsc{exit}_X; \textsc{exit}_X$ does not contribute anything to the composition. However, this is correct since X is not nullable, and the second $\textsc{exit}_X$ is not enabled after the counter value of X is reset to 0. Intuitively, the second occurrence of X cannot be exited without iterating X at least once.

$$
\begin{aligned}
\partial_\alpha(\varepsilon) &\stackrel{\text{def}}{=} \emptyset \\
\partial_\alpha(\psi Z) &\stackrel{\text{def}}{=} \begin{cases} \{\langle \textsc{id}, Z\rangle\} & \text{if } \alpha \wedge \psi \text{ is satisfiable} \\ \emptyset & \text{otherwise} \end{cases} \\
\partial_\alpha((W|Y)Z) &\stackrel{\text{def}}{=} \partial_\alpha(WZ) \cup \partial_\alpha(YZ) \\
\partial_\alpha(S{*}Z) &\stackrel{\text{def}}{=} \partial_\alpha(S) \otimes \{\langle \textsc{id}, S{*}Z\rangle\} \cup \partial_\alpha(Z) \\
\partial_\alpha(XZ) &\stackrel{\text{def}}{=} \partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\} \cup \\
&\qquad \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_\alpha(Z)
\end{aligned}
$$

Example 5.2. Consider the regex $R = \texttt{.*a\{1,3\}a\{1,3\}a}$. Let X be the counting loop $\texttt{a\{1,3\}}$. R has two minterms a and $\texttt{[\^{}a]}$. We get the following conditional derivatives of R, starting with the case for $\partial_\alpha(S{*}Z)$ due to the normal form assumption:

$$
\begin{aligned}
\partial_{\mathsf{a}}(R) &= \partial_{\mathsf{a}}(\texttt{.}) \otimes \{\langle \textsc{id}, R\rangle\} \cup \partial_{\mathsf{a}}(XX\mathsf{a}) \\
&= \{\langle \textsc{id}, R\rangle, \langle \textsc{incr}_X, XX\mathsf{a}\rangle, \langle \textsc{exit1}_X, X\mathsf{a}\rangle\} \\
\partial_{\mathsf{a}}(XX\mathsf{a}) &= \partial_{\mathsf{a}}(\mathsf{a}) \otimes \{\langle \textsc{incr}_X, XX\mathsf{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_{\mathsf{a}}(X\mathsf{a}) \\
&= \{\langle \textsc{incr}_X, XX\mathsf{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \{\langle \textsc{incr}_X, X\mathsf{a}\rangle, \langle \textsc{exit}_X, \varepsilon\rangle\} \\
&= \{\langle \textsc{incr}_X, XX\mathsf{a}\rangle, \langle \textsc{exit1}_X, X\mathsf{a}\rangle\} \\
\partial_{\mathsf{a}}(X\mathsf{a}) &= \partial_{\mathsf{a}}(\mathsf{a}) \otimes \{\langle \textsc{incr}_X, X\mathsf{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_{\mathsf{a}}(\mathsf{a}) \\
&= \{\langle \textsc{incr}_X, X\mathsf{a}\rangle, \langle \textsc{exit}_X, \varepsilon\rangle\} \\
\partial_{\mathsf{a}}(\mathsf{a}) = \partial_{\mathsf{a}}(\texttt{.}) = \partial_{\texttt{[\^{}a]}}(\texttt{.}) &= \{\langle \textsc{id}, \varepsilon\rangle\} \\
\partial_{\texttt{[\^{}a]}}(\mathsf{a}) &= \emptyset
\end{aligned}
$$

Above, the composition $\textsc{exit}_X; \textsc{exit}_X$ in $\partial_{\mathsf{a}}(XX\mathsf{a})$ is undefined and thus removed. We also get that $\partial_{\texttt{[\^{}a]}}(R) = \{\langle \textsc{id}, R\rangle\}$ where $\partial_{\texttt{[\^{}a]}}(\mathsf{a}) = \emptyset$ and consequently $\partial_{\texttt{[\^{}a]}}(XX\mathsf{a}) = \emptyset$ and $\partial_{\texttt{[\^{}a]}}(X\mathsf{a}) = \emptyset$.

If we now consider, for example, the language defined by $\partial_{\mathsf{a}}(X\mathsf{a})$ in a valid counter memory $\mathfrak{m}$, it is the union of the languages $\mathrm{L}^{\textsc{incr}_X(\mathfrak{m})}(X\mathsf{a})$ and $\mathrm{L}^{\textsc{exit}_X(\mathfrak{m})}(\varepsilon)$. These correspond to the cases of continuing to iterate the loop X (if the counter value of X is below 3) or exiting the loop (if the counter value of X is at least 1) and accepting $\{\epsilon\}$. □

Example 5.3. Consider the regex $\texttt{(.\{9\})*}$, whose CA is in Fig. 2. Here, . is the only input predicate and denotes the set of all characters. We explain the use of some of the counter operations in the CA of Fig. 2 by showing how they arise through the partial-derivative-based construction of CAs as discussed above. The initial state is the regex itself. The (only) partial derivative of the state $\texttt{(.\{9\})*}$ is $\texttt{.\{9\}(.\{9\})*}$ where the body of the counting loop is exited but also incremented once, so EXIT1 is applied to c under the guard $\textsc{canExit}_c$ (which is shown as $c \geq 9/c{:=}1$ in the figure). The state $\texttt{.\{9\}(.\{9\})*}$ has two cases of partial derivatives both leading back to $\texttt{.\{9\}(.\{9\})*}$.

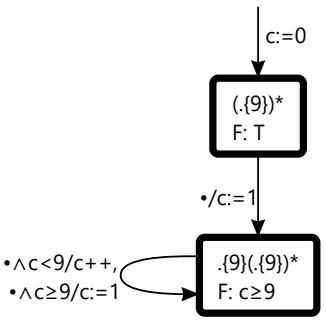


Fig. 2. CA($\texttt{(.\{9\})*}$)

The first case is when $c < 9$ ($\textsc{canIncr}_c$ holds), in which case c is incremented (shown as $c < 9/c{+}{+}$ in the figure). The second case is when the counting loop is conditionally nullable and is exited under the condition $\textsc{canExit}_c$ (i.e. $c \geq 9$), the value of c is reset to 0, and then c is

incremented as a result of taking the partial derivative of `(.{9})*`. Thus, EXIT1 arises as a sequential composition of exiting the loop, followed by resetting the counter to 0, and then incrementing it. Therefore, $\textsc{canExit}_c$ must hold, while the increment condition holds trivially after a reset to 0. The initial state is unconditionally final in Fig. 2, while the other state is final only when $\textsc{canExit}_c$ holds as marked by "F :". □

We now state the correctness theorem of conditional derivatives. For that, we define $\textsc{canExit}_R$ as the predicate shown above for a normalized regex R, assuming that X stands for a counting loop.

$$\textsc{canExit}_R \stackrel{\text{def}}{=} \begin{cases} \top_{\mathbb{C}} & \text{if } R = \varepsilon, \\ \textsc{canExit}_Z & \text{else if } R = YZ \text{ and } Y \text{ is nullable}, \\ \textsc{canExit}_X \wedge \textsc{canExit}_Z & \text{else if } R = XZ, \\ \bot_{\mathbb{C}} & \text{otherwise.} \end{cases}$$

Note that Y above may also be a counting loop. However, since it is nullable, $\boldsymbol{min}_Y$ must be 0, and then $\textsc{canExit}_Y$ is always true. (If $\boldsymbol{min}_Y > 0$, then Y cannot be nullable as R is normalized.)

We further need the following additional notions too. A counter X *is visible in* R if either $R = YZ$ and $X = Y$, or else if X does not occur in Y and X is visible in Z. A counter memory $\mathfrak{m}$ is *valid for* R if $\mathfrak{m}(X) = 0$ for all invisible counters X that occur in R. Correctness of the construction of conditional derivatives is stated in Theorem 5.4—see [Turoňová et al. 2020] for a detailed proof.

THEOREM 5.4. *Let R be a normalized regex and let $\Sigma = \mathit{Minterms}(\Theta)$ where Θ is some finite superset of $\mathit{Preds}(R)$. If $\mathfrak{m}$ is valid for R, then* $\mathrm{L}^{\mathfrak{m}}(R) = \bigcup_{\alpha \in \Sigma} [\![\alpha]\!] \cdot \mathrm{L}^{\mathfrak{m}}(\partial_\alpha(R)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_R\}$.

5.3 Constructing CAs from Conditional Derivatives

We convert a normalized regex R to the counting automaton $\mathrm{CA}(R)$ whose set of states is the smallest set containing R as the initial state and all those regexes that arise in conditional derivatives constructed from R by repeated derivation wrt Σ. Given a state represented by a regex S, for each $\alpha \in \Sigma$ and each partial conditional derivative $\langle f, T\rangle \in \partial_\alpha(S)$, there is a transition $S \text{-}(\alpha,f)\!\!\rightarrow T$ in $\mathrm{CA}(R)$. The *final condition* $F(S)$ of a state S of $\mathrm{CA}(R)$ is $\textsc{canExit}_S$. Observe that $F(S) = \bot_{\mathbb{C}}$ when S is not nullable and has no visible counters, which corresponds to the classical case.

As shown in [Turoňová et al. 2020] the following result can be proved using Theorem 5.4.

THEOREM 5.5. *Let R be a normalized regex and $A = \mathit{FA}(\mathit{CA}(R))$. Then, for all $\langle \mathfrak{m}, S\rangle \in Q_A$, $\mathcal{L}_A(\langle \mathfrak{m}, S\rangle) = \mathrm{L}^{\mathfrak{m}}(S)$.*

The construction of $\mathrm{CA}(R)$ terminates, and the number of states of $\mathrm{CA}(R)$ is linear in $\sharp_\Psi(R)$.

THEOREM 5.6. *Let R be a normalized regex. Then $|Q_{CA(R)}| \leq \sharp_\Psi(R) + 1$.*

A proof of Theorem 5.6 is in [Turoňová et al. 2020]. We get the following final correctness result as a corollary of Theorem 5.5, Theorem 5.1, and Theorem 5.6.

COROLLARY 5.7. *Let R be a normalized regex. Then $\mathcal{L}(R) = \mathcal{L}(\mathit{CA}(R))$.*

PROOF. First, $Q_{\mathrm{CA}(R)}$ is finite, and thus well-defined by using Theorem 5.6. Use Theorem 5.5 with $\langle \mathfrak{m}, S\rangle$ as the initial state $\langle 0, R\rangle$ of A. It follows that $\mathcal{L}(A) = \mathrm{L}^0(R)$. Then use Theorem 5.1 for $\mathrm{L}^0(R) = \mathcal{L}(R)$ and $\mathcal{L}(\mathrm{CA}(R)) = \mathcal{L}(A)$ holds by definition. □

A further important aspect of $\mathrm{CA}(R)$ is that, although the number of input minterms may potentially be exponential in the number of predicates in R, in the case of predicates being represented as a finite union of intervals (as is done typically for character classes), the size of a single predicate representation can be estimated to be proportional to the number of interval borders in the union. In this case, the total size of all the minterms remains linear in the total size of all the predicates

because the total number of interval borders will remain the same in minterms as in the original set of predicates. In other words, mintermization based on character classes does not blow up the number of transition in CA(R). We have also validated this fact experimentally.

6 FROM COUNTING AUTOMATA TO COUNTING-SET AUTOMATA

CAs obtained through conditional derivatives as shown in the previous section are nondeterministic. As one of the main contributions of this work, we now propose an approach for determinizing them into a form that can be used efficiently for regex matching. The approach from which we start and to which we contrast our new method is the naive determinization of CAs to DFAs: The given CA is first converted to its underlying NFA, by making the counter memories an explicit part of control states. The NFA is in turn determinized by the textbook subset construction.

Already the first step, the construction of the NFA, oftentimes explodes since it sacrifices the succinctness of symbolic counters (it is linear to the counter bounds). This initial blow-up is then much amplified in the subset construction, which is exponential to the size of the NFA and hence also to the counter bounds (as, e.g., in the case of the regex `.*a.{`k`}` with its CA in Fig. 1a).

Our answer to this problem is a direct determinization of the CA into a novel type of automata, which we call *counting-set automata (CsAs)*. Control states of counting-set automata produced by our determinization are essentially the states of the corresponding DFA but with the counter memories removed. In order to be able to simulate a run of the DFA, they are equipped with special registers that can hold *sets* of integers, and they use them to compute the right counter memories at runtime. This completely avoids the state space explosion of the naive construction caused by wiring counter memories into control states. Moreover, the simulation is fast because all the manipulations with a counting set can be done in constant time.

6.1 Counting-Set Automata

We now formalize the idea of counting-set automata outlined above. We use the notion of a combined Boolean algebra $\mathbb{I}\times\mathbb{S}$, which allows us to manipulate pairs of predicates from the input algebra $\mathbb{I}$ and the counting-set algebra $\mathbb{S}$. For the purposes of this paper, we assume that predicates in $\Psi_{\mathbb{I}\times\mathbb{S}}$ have the form $\alpha \wedge \beta$ where $\alpha \in \Psi_{\mathbb{I}}$ and $\beta \in \Psi_{\mathbb{S}}$. The conjuction $(\alpha \wedge \beta) \wedge_{\mathbb{I}\times\mathbb{S}} (\alpha' \wedge \beta')$ has the usual meaning of $(\alpha \wedge_{\mathbb{I}} \alpha') \wedge (\beta \wedge_{\mathbb{S}} \beta')$ and $\alpha \wedge \beta$ is satisfiable if both α and β are satisfiable in their respective algebras.

Counting sets. We consider a set-based interpretation of counters where the value of a counter c is a *finite set* rather than a single value. A counter under such an interpretation is referred to as a *counting set*. A *(counting-)set memory* for C is a function $\mathfrak{s} : C \to \mathcal{P}_{\text{fin}}(\mathbb{N})$ such that, for all $c \in C$, $\mathit{Max}(\mathfrak{s}(c)) \leq \boldsymbol{max}_c$.[3] Observe that the set of all set memories for C is *finite*. Counting-set predicates over C form an effective Boolean algebra $\mathbb{S}_C$ called the *counting-set algebra over* C, also denoted just $\mathbb{S}$ when C is clear from the context, whose domain $\mathfrak{D}_{\mathbb{S}}$ is the set of all set memories for C. The set of predicates $\Psi_{\mathbb{S}}$ is the Boolean closure of the basic predicates $\textsc{canIncr}_c$ and $\textsc{canExit}_c$, hence syntactically the same as in the counter algebra $\mathbb{C}$, but with a different semantics under $\mathbb{S}$:

$$\mathfrak{s} \models \textsc{canExit}_c \iff \mathit{Max}(\mathfrak{s}(c)) \geq \boldsymbol{min}_c \quad \text{and} \quad \mathfrak{s} \models \textsc{canIncr}_c \iff \mathit{Min}(\mathfrak{s}(c)) < \boldsymbol{max}_c$$

where $\mathit{Min}(\cdot)$ and $\mathit{Max}(\cdot)$ are the set minimum and maximum, respectively. Intuitively, the conditions test existence of a set element satisfying the same counter condition.

Counting-set automata. A *counting-set automaton* (CsA) is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ where: $\mathbb{I}$ is an effective Boolean algebra called the *input algebra*. C is a finite set of *counters* associated with the counting-set algebra $\mathbb{S}$. Q is a finite set of *states* with $q_0 \in Q$ being the *initial state*.

[3] We write $\mathcal{P}_{\text{fin}}(X)$ for the powerset of X restricted to finite nonempty sets.

$F : Q \to \Psi_{\mathbb{S}}$ is the *final-state condition*. $\Delta \subseteq Q \times \Psi_{\mathbb{I}\times\mathbb{S}} \times (C \to \mathcal{P}(O)) \times Q$ is a finite set of *transitions*. The second component is its *guard*. The third component is the *counting-set operator* in which $O = \{\textsc{Incr}, \textsc{Noop}, \textsc{Rst}, \textsc{Rst1}\}$ is the set of *counting-set operations*. They are essentially counter operations lifted to sets (note the use of the larger initial letters to distinguish them from the counter operations). We also use the different names Rst and Rst1 for the lifting of exit and exit1 to stress their different usage (not only for exiting a loop but also for initialisation when entering the loop as will become clear in Eq. (7)). Sets of counting-set operations assigned to every counter by the counting-set operator are called *combined (counting-set) operations*.

The CsA A is *deterministic* iff the following holds for every two transitions $p \text{-}(\psi_1,f_1)\!\!\to q_1$ and $p \text{-}(\psi_2,f_2)\!\!\to q_2$ in Δ: if $\psi_1 \wedge \psi_2$ is satisfiable, then $f_1 = f_2$ and $q_1 = q_2$.

Semantics of CsAs. The semantics of an indexed counting-set operation $\textsc{op}_c \in O$ is the set transformer $upd(\textsc{op}_c)$ defined as follows:

$$upd(\textsc{Incr}_c) = \boldsymbol{\lambda} S.\{n+1 \mid n \in S \wedge n < \boldsymbol{max}_c\} \qquad upd(\textsc{Rst}_c) = \boldsymbol{\lambda} S.\{0\}$$
$$upd(\textsc{Noop}_c) = \boldsymbol{\lambda} S.S \qquad upd(\textsc{Rst1}_c) = \boldsymbol{\lambda} S.\{1\}$$

Then, the counting-set operator $f : C \to \mathcal{P}(O)$ is assigned the counting-set-memory transformer $\mathrm{f} : \mathfrak{D}_{\mathbb{S}} \to \mathfrak{D}_{\mathbb{S}}$ defined as follows:

$$\mathrm{f}(\mathfrak{s}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} c. \begin{cases} \bigcup_{\textsc{op} \in f(c)} upd(\textsc{op}_c)(\mathfrak{s}(c)) & \text{if } f(c) \neq \emptyset \\ \{0\} & \text{if } f(c) = \emptyset \end{cases}$$

That is, (1) if $f(c) \neq \emptyset$, then the value $\mathfrak{s}(c)$ of each counting set c is transformed into the union of the counting sets that result from applying the operations from $f(c)$ on $\mathfrak{s}(c)$, and (2) if $f(c) = \emptyset$, then c is implicitly reset to $\{0\}$ (an implicit Rst). Our determinization procedure creates such transitions when the value of c is irrelevant (when c is a dead variable).

Note that, unlike counter operators of a CA, a counting-set operator f does not induce any guard. The guard is rather a separate component of the transition. This is because CsA transitions produced in the CA-to-CsA determinization need guards that are partially independent of the operations of f. In particular, we will we need to distinguish cases such as $\neg\textsc{canExit}_c \wedge \textsc{canIncr}_c$, $\textsc{canExit}_c \wedge \neg\textsc{canIncr}_c$, or $\textsc{canExit}_c \wedge \textsc{canIncr}_c$. The guard hence cannot be induced by f alone.

Note also that, unlike in CAs, the updates are defined for *indexed* operations. The reason is that the semantics of the Incr operation is restricted to never produce values greater than $\boldsymbol{max}_c$.

Finally, the *language of the CsA A* is defined through its underlying *configuration FA*, $FA(A)$, as $\mathcal{L}(A) := \mathcal{L}(FA(A))$. The states of $FA(A)$ are *configurations* of A, namely, tuples of the form $(q, \mathfrak{s}) \in Q \times \mathfrak{D}_{\mathbb{S}}$ consisting of a state q and a counting-set memory $\mathfrak{s}$. There are finitely many such configurations. The initial state of $FA(A)$ is the *initial configuration* $(q_0, \{c \mapsto \{0\}\}_{c\in C})$ of A. A transition $\tau = p \text{-}(\alpha\wedge\beta,f)\!\!\to q \in \Delta$ is *enabled* in a configuration $(p, \mathfrak{s})$ iff α is satisfiable and $\mathfrak{s} \in [\![\beta]\!]_{\mathbb{S}}$, meaning that $\mathfrak{s}$ satisfies the counter guard β. If τ is enabled in $(p, \mathfrak{s})$, then $FA(A)$ contains the transition $(p, \mathfrak{s}) \text{-}(\alpha)\!\!\to (q, \mathrm{f}(\mathfrak{s}))$. Finally, a state $(q, \mathfrak{s})$ of $FA(A)$ is *final* iff $\mathfrak{s} \models F(q)$.

Example 6.1. An example of a CsA is in Fig. 1b. It uses intuitive notations that were also introduced in Section 2 as abbreviations for the operations of the counting-set data structure. Counting-set operators are depicted as assignments to c, Rst is represented as $\{0\}$ on the right of the assignment, Rst1 is represented by $\{1\}$, Incr by $c + 1$, and Noop by c. Multiple transitions between the same states and with the same updates are merged into one with a simplified guard. An example whose notation closely follows the formal development is in Fig. 3. □

Runtime efficiency of counting sets. A major reason for choosing CsAs as the target kind of machine for determinization of CAs is that pattern matching with CsAs is fast. Using the data structure explained in Section 2, all the basic counting-set tests and updates, namely, $\textsc{canIncr}_c$,

canExit$_c$, Noop, Incr, Rst, and Rst1, can be implemented to run in constant time regardless of the size of the counting set and the value $\boldsymbol{max}_c$ (assuming constant-time complexity of integer arithmetic operations). Moreover, almost all combined counting-set operations can be implemented to run in constant time too. In particular, when at most one counting-set operation of a given combined operation returns a set other than $\{0\}$ or $\{1\}$, their union can be computed in constant time. However, the union of two general sets, other than $\{0\}$ and $\{1\}$, would take time linear to the size of the sets (which is at most $\boldsymbol{max}_c$). The only operations that may return sets other than $\{0\}$ or $\{1\}$ are Noop and Incr. We therefore call a transition *slow* if its counting-set operator f assigns to some counter c the result of a combined operation $f(c)$ that contains both Noop and Incr. A CsA that has slow transitions is called *slow*, and a CsA that does not have them is called *fast*. Slow CsAs are fortunately rare in practice (cf. Section 7).

When a fast CsA is used in pattern matching, tests and updates of one counting set then take $O(1)$ time, which in turn gives $O(|C|)$ for all counting sets and their unions. *This is our major achievement: the independence of the running time from the counter bounds.*

6.2 Encoding DFA Powerstates as CsA Configurations

In order to build intuition needed for understanding our determinization algorithm, we will first concretize how the configurations of a CsA can encode states of a DFA corresponding to the NFA $FA(A)$ underlying a given CA $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$. First, recall that, since A is converted into $FA(A)$ by making the counter memories explicit parts of control states, the states of $FA(A)$ are pairs $(p, \mathfrak{m})$ consisting of a state p of A and a counter memory $\mathfrak{m}$. Second, assume that $FA(A)$ is determinized using the textbook subset construction.[4] We denote the result as $DFA(A)$ from now on. Then, the states of $DFA(A)$ are sets of states of $FA(A)$, i.e., sets of pairs $(p, \mathfrak{m})$, which we will call *powerstates*. The control states of the CsA A' built by our CA-to-CsA determinization will be subsets of the set Q of states of the CA A. The configurations of A' will thus be pairs $(R, \mathfrak{s})$ where $R \subseteq Q$ is a CsA control state, i.e., a set of states of A, and $\mathfrak{s} : C \to \mathcal{P}_{\text{fin}}(\mathbb{N})$ is a counting-set memory. Let us now consider how $\mathfrak{s}$ can be interpreted in this context.

Naive encoding. A naive interpretation of a CsA configuration $(R, \mathfrak{s})$ is a DFA state containing all pairs $(r, \mathfrak{m})$ such that $r \in R$ and, for all $c \in C$, $\mathfrak{m}(c)$ can be any value from $\mathfrak{s}(c)$. The set of the counter memories $\mathfrak{m}$ is then isomorphic to the Cartesian product $\prod_{c \in C} \mathfrak{s}(c)$ of the sets $\mathfrak{s}(c)$ assigned to the counters, and the entire powerstate is the Cartesian product $R \times \mathfrak{m}$ of the set of states and the set of counter memories. The naive interpretation, however, is too impractical as it cannot express any dependence of a counter memory on the CA state (every state can be paired with each considered memory) nor any mutual dependence of values of different counters within a counter memory (every possible value of a counter c can be paired with every possible value of any other counter d). Most DFAs compiled from real-life regexes do not fit into this representation. For instance, the DFA configuration $\{(q, c = 0), (s, c = 0), (s, c = 1)\}$ of the CA from Fig. 1 in Section 2 could not be represented by a CsA configuration because q and s appear with different sets of counter values.

Encoding with counter scopes. Our key observation how to resolve the above problem (at least for many real-life scenarios) is to take advantage of that not every counter is "used" at every CA state. In fact, the value of a counter is usually implicitly 0 at most states except a few. If these states are known, the implicit zeros do not have to be remembered explicitly in the counting sets, and the encoding becomes much more flexible. To formalize this, we introduce the notion of the *scope of a*

[4] The DFA produced by the textbook subset construction from a *simple* FA $\mathcal{A} = (\mathbb{I}, Q, q_0, F, \Delta)$ will have $\mathcal{P}(Q)$ as the set of states, transitions $S \xrightarrow{(\alpha)} \{r \in Q \mid s \xrightarrow{(\alpha)} r \in \Delta, s \in S\}$, the initial state $\{q_0\}$, and as the final states all those intersecting F. We note that to determinize a CA which is not simple, one could start from the more sophisticated version of the subset construction for symbolic automata of [Veanes et al. 2010], which avoids explicit generation of all minterms.

counter that over-approximates the set of states where a counter c can have a non-zero value and that is easy to compute.[5] The scope is defined inductively as the smallest set of states $\sigma(c)$ such that

(1) $q \in \sigma(c)$ if there is a transition to q with either $\textsc{incr}_c$ or $\textsc{exit1}_c$, or
(2) there is a transition to q from a state in $\sigma(c)$ with the $\textsc{noop}_c$ operation.

In other words, the scope of c spreads from an increment of c along the transition relation until a transition with $\textsc{exit}_c$.

The DFA *powerstate encoded by a CsA configuration* $(R, \mathfrak{s})$ can then be formally defined as the set $(R, \mathfrak{s})^{DFA}$ of configurations $(r, \mathfrak{m})$ of the CA A such that $r \in R$ and, for all $c \in C$, $\mathfrak{m}(c) \in \mathfrak{s}(c)$ if $c \in \sigma(r)$, else $\mathfrak{m}(c) = 0$. We call the powerstates of $DFA(A)$ that can be encoded by CsA configurations *Cartesian*, and call the entire DFA Cartesian if all its powerstates are Cartesian.

Example 6.2. The powerstates of the $DFA(A)$ of the CA A from Fig. 1a are indeed Cartesian (as discussed in Section 2) because q_0 is not in the scope of c. The encoding of powerstates by CsA configurations is also illustrated in Section 2 and later also in Example 6.4. □

The Cartesian encoding still cannot express all kinds of DFA powerstates. In particular, it cannot express more subtle dependencies of counter values on the state, and dependencies of counter values of different counters on each other, which mainly concerns CAs with nested counting loops compiled from regexes with nested counting sub-expressions. Example 6.5 discusses a regex that leads to a non-Cartesian CA. However, we later present a strong empirical evidence that a significant majority of real-life regexes lead to Cartesian CA.

6.3 Generalized Subset Construction

We will now describe the core of our CA-to-CsA determinization. It is built on top of the textbook subset construction for NFAs. We use the CA from Fig. 3a as a running example through the section. We make a simplifying assumption that the input CAs are simple (different character classes on their transitions do not overlap). This is satisfied by CAs generated by the derivative construction from Section 5 since their transitions are labeled by minterms of the original regex. The assumption could be dropped and the construction could be relatively easily generalized in the style of symbolic automata determinization of [Veanes et al. 2010].

Let $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ be a simple CA with the scope function $\sigma : Q \to \mathcal{P}(C)$. The algorithm produces the deterministic CsA $A' = (\mathbb{I}, C, Q', S_0, F', \Delta')$ whose components are constructed as described below. Namely, control states of A', called powerstates, are subsets of Q, i.e., $Q' \subseteq \mathcal{P}(Q)$. The initial powerstate is $S_0 = \{q_0\}$. A powerstate $S \in Q'$ is final iff the final condition holds for some of its elements, i.e., $F'(S) \stackrel{\text{def}}{=} \bigvee_{q \in S} F(q)$. The sets Δ' and Q' are constructed by a fixpoint computation that explores the state space reachable from S_0. During the construction, transitions starting from previously reached powerstates are constructed and included together with their target states into Δ' and Q', respectively, until no new powerstates can be reached.

Transitions starting from a given control state R of the CsA A' are constructed to update the runtime values of counting sets such that they simulate transitions of the DFA corresponding to the CA A. Assume a CsA configuration $(R, \mathfrak{s})$ and a DFA transition $(R, \mathfrak{s})^{DFA} \text{-}(\alpha)\!\!\rightarrow P$ from the DFA powerstate encoded by $(R, \mathfrak{s})$ over an input minterm α. The simulating CsA transition must transform $(R, \mathfrak{s})$ into $(R', \mathfrak{s}')$ with $(R', \mathfrak{s}')^{DFA} = P$. The simulated DFA transition was constructed from α-transitions of the NFA $FA(A)$ that are actually instantiations of the CA α-transitions enabled

[5] Computing the precise set of states where a counter c can have a non-zero value would require a reachability analysis in the general case (since some of the transitions may never be executable—think of simultaneously counting with counters c and d such that $\textsc{canIncr}_c < \textsc{canExit}_d$, then the exit transition for d will never be taken). For CAs produced by our derivative construction, the scope, however, corresponds to this set precisely—no transitions that are never executable are generated.

in configurations $(r, \mathfrak{m}) \in (R, \mathfrak{s})^{DFA}$. The simulating CsA transition will be constructed from these CA transitions. They can be identified by (1) their source state, which must be in R, (2) an alphabet minterm $\alpha \in \Sigma$ where Σ is the set of minterms over all input predicates in the CA A, and (3) their compatibility with a particular set of enabled/disabled counter guards. This set of guards belongs to the set of minterms $\Gamma_{R,\alpha}$ of the set of counter guards on the α-transitions originating in R:

$$\Gamma_{R,\alpha} \stackrel{\text{def}}{=} \mathit{Minterms}(\{\mathit{grd}(\text{OP}_c) \mid r \text{-}(\alpha,f)\!\!\rightarrow s \in \Delta, r \in R \wedge c \in \sigma(r), \text{OP}_c \in f\}).$$

Hence, the CsA will have a transition leaving R for each $\alpha \in \Sigma$ and $\beta \in \Gamma_{R,\alpha}$, and the transition will be built from the set of CA α-transitions originating in R and *consistent* with β:

$$\Delta_{R,\alpha,\beta} \stackrel{\text{def}}{=} \{r \text{-}(\alpha,f)\!\!\rightarrow s \in \Delta \mid r{\in}R, \boldsymbol{\mathit{Sat}}(\varphi_f \wedge \beta)\}.$$

Its target is the set T of all target states of the transitions in $\Delta_{R,\alpha,\beta}$, and its guard is $\alpha \wedge \beta$.[6]

The remaining component is the counting-set operator f'. It must summarize the updates of the counter values on transitions of $\Delta_{R,\alpha,\beta}$ as updates of the respective counting sets. The values of counters that are out of scope, hence implicitly zero, will not be tracked in counting sets. Tracking the value of a counter hence starts when A' simulates a transition of A entering the scope of the counter, and ends when no state from the scope is present in the target CsA state.

Let $\Delta_{R,\alpha,\beta}(c)$ be the set of transitions in $\Delta_{R,\alpha,\beta}$ with the target state in the scope of c. The counting-set operator f' is built in the form $f'(c) \stackrel{\text{def}}{=} \{\mathit{op}(\tau, c) \mid \tau \in \Delta_{R,\alpha,\beta}(c)\}$. Here, $\mathit{op}(\tau, c)$ denotes the counting-set operation that, given a CA transition $\tau = p \text{-}(\alpha,f)\!\!\rightarrow q$, transforms the set of possible values of the counter c at the state p to the set of values obtained at q after taking the transition. It is defined in Eq. (7) on the right. The set operation induced by the CA transition corresponds to the counter operation on the transition. In the third and fourth case, when the CA transition comes from out of the scope, it is certain that the counter can only have the value 0, which is the same value as produced by EXIT (or EXIT1 when the counter is immediately incremented). The resulting CsA transition is therefore $S \text{-}(\alpha\wedge\beta,f')\!\!\rightarrow T$. Note that $f'(c)$ ends up empty when the target powerstate is fully out of the scope of c, which semantically corresponds to the implicit reset to $\{0\}$.

$$\mathit{op}(p \text{-}(\alpha,f)\!\!\rightarrow q, c) \stackrel{\text{def}}{=} \begin{cases} \text{NOOP} & \text{if } f(c) = \text{NOOP} \wedge p \in \sigma(c) \\ \text{INCR} & \text{if } f(c) = \text{INCR} \wedge p \in \sigma(c) \\ \text{RST} & \text{if } f(c) = \text{NOOP} \wedge p \notin \sigma(c) \\ \text{RST1} & \text{if } f(c) = \text{INCR} \wedge p \notin \sigma(c) \\ \text{RST} & \text{if } f(c) = \text{EXIT} \\ \text{RST1} & \text{if } f(c) = \text{EXIT1} \end{cases} \tag{7}$$

Observe that A' is deterministic since, for any two distinct transitions $S \text{-}(\alpha_1,f_1)\!\!\rightarrow S_1$ and $S \text{-}(\alpha_2,f_2)\!\!\rightarrow S_2$, the condition $\alpha_1 \wedge \alpha_2$ is unsatisfiable by virtue of minterms.

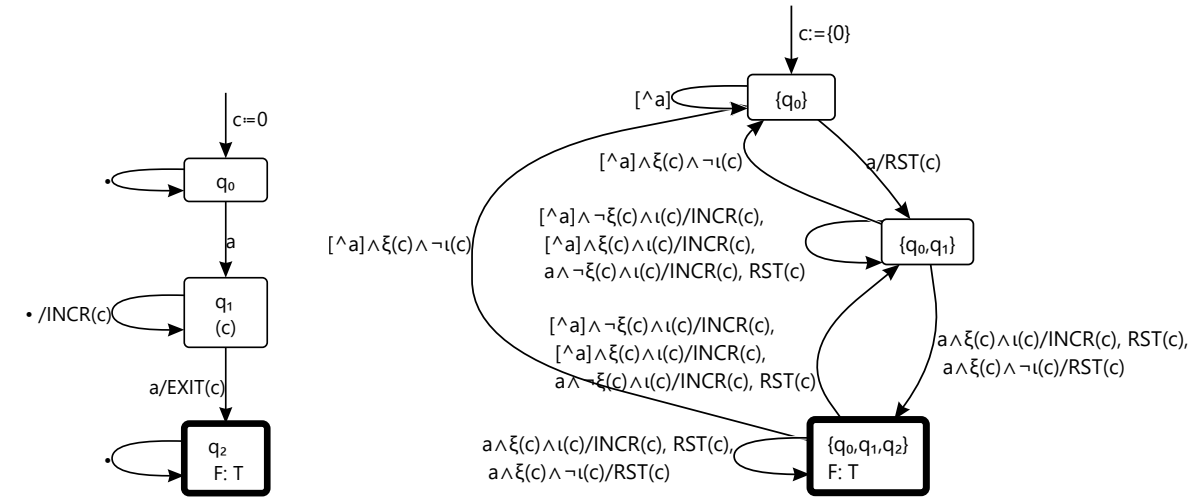
THEOREM 6.3. *For the CA A and the CsA A' above, we have $\mathcal{L}(A') \supseteq \mathcal{L}(A)$ and $|Q'| \leq 2^{|Q|}$.*

PROOF (IDEA). The language inclusion is proved by showing that the configuration automaton $FA(A')$ of A' simulates $DFA(A)$, more concretely, that each configuration $(R, \mathfrak{s})$ of A', a state of $FA(A')$, simulates the powerstate $(R, \mathfrak{s})$ of $DFA(A)$. The bound on the size of the state space follows from that states of the CsA are sets of states of the CA. □

Example 6.4. Consider the CA in Fig. 3a that has states q_0, q_1, and q_2. The state q_0 is initial, the final condition of q_2 is $\top$, and it is $\bot$ for q_0 and q_1. The set of counters is $C = \{c\}$ with $\sigma(c) = \{q_1\}$ (i.e., c is not used and hence implicitly 0 in q_0 and q_2). Finally, $\Sigma = \{\mathtt{a}, \mathtt{[\^{}a]}\}$. In Fig. 3a, we compactly represent transitions over all minterms from Σ using $\boldsymbol{.}$. The determinization starts exploring the CsA from its initial state $S_0 = \{q_0\}$.

Let us focus on the transitions for the input minterm $\alpha = \mathtt{a}$. Two transitions are leaving q_0, namely $\delta_1 = q_0 \text{-}(\mathtt{a},\text{NOOP}_c)\!\!\rightarrow q_0$ and $\delta_2 = q_0 \text{-}(\mathtt{a},\text{NOOP}_c)\!\!\rightarrow q_1$, both with no guard on c, hence $\Gamma_{S_0,\alpha} = \{\top\}$. The

[6] Recall that the predicates in $\Psi_{\mathbb{C}}$ and $\Psi_{\mathbb{S}}$ are syntactically the same.



(a) The CA for `.*a.{4,8}a` (b) Determinization of the CA into a CsA

Fig. 3. From a regex via a CA to a deterministic CsA. We are using a notation closely following the formal development. We only use $\text{OP}(c)$ instead of OP_c and abbreviate CANEXIT_c by $\xi(c)$ and CANINCR_c by $\iota(c)$.

guard $\top$ is thus the only choice for the counter minterm β. The set $\Delta_{R,\alpha,\beta}$ of transitions consistent with α and β then contains both a-transitions δ_1 and δ_2 originating from q_0. Since δ_2 is entering the scope of c, it generates the counting-set operation RST_c according to the third case of Eq. (7). Since δ_1 stays out of the scope, it does not generate any counting-set operations. We obtain the counting-set operator $f' = \{\text{RST}_c\}$ and generate the CsA transition $\tau_1 = \{q_0\} \text{-(a}\wedge\beta, \{\text{RST}_c\}\text{)}\!\!\rightarrow \{q_0, q_1\}$.

Next, let us focus on the a-transitions from $S_1 = \{q_0, q_1\}$. Here, $\Gamma_{S_1,\text{a}}$ has the following three satisfiable elements: $\text{CANEXIT}_c \wedge \text{CANINCR}_c$, $\neg\text{CANEXIT}_c \wedge \text{CANINCR}_c$, and $\text{CANEXIT}_c \wedge \neg\text{CANINCR}_c$ (the guard $\neg\text{CANEXIT}_c \wedge \neg\text{CANINCR}_c$ is excluded as it is never satisfied for non-empty sets of positive integers). Let us generate a transition for the second case, $\beta = \neg\text{CANEXIT}_c \wedge \text{CANINCR}_c$. We obtain $\Delta_{S_1,\text{a},\beta} = \{q_0 \text{-(a,}\text{NOOP}_c\text{)}\!\!\rightarrow q_0,\ q_0 \text{-(a,}\text{NOOP}_c\text{)}\!\!\rightarrow q_1,\ q_1 \text{-(a,}\text{INCR}_c\text{)}\!\!\rightarrow q_1\}$. As before, the first transition does not contribute to f' as it stays out of the scope, and the second transition adds RST_c. The third transition adds INCR_c (the second case of Eq. (7)). The resulting CsA transition is thus $\tau_2 = S_1 \text{-(a}\wedge\neg\text{CANEXIT}_c\wedge\text{CANINCR}_c, \{\text{INCR}_c, \text{RST}_c\}\text{)}\!\!\rightarrow S_1$. The rest of the construction is analogous.

Last, let us also illustrate the simulation of $DFA(A)$ by the constructed CsA transitions. On the word aa, the DFA would execute the run $\{(q_0, c = 0)\} \text{-(a)}\!\!\rightarrow \{(q_0, c = 0), (q_1, c = 0)\} \text{-(a)}\!\!\rightarrow \{(q_0, c = 0), (q_1, c = 0), (q_1, c = 1)\}$. The simulating run of our CsA would start in the initial configuration $\{\{q_0\}, c \in \{0\}\}$. The transition τ_1 would produce the configuration $\{\{q_0, q_1\}, c \in \{0\}\}$ (since $\text{RST}(\{0\}) = \{0\}$) from where τ_2 would produce $\{\{q_0, q_1\}, c \in \{0, 1\}\}$ (since $\text{INCR}(\{0\}) = \{1\}$ and $\text{RST}(\{0\}) = \{0\}$). The sequence of configurations precisely encodes the sequence of the DFA powerstates, that is, the sequnce $(\{q_0\}, c \in \{0\})^{DFA} = \{(q_0, c = 0)\}$; $(\{q_0, q_1\}, c \in \{0\})^{DFA} = \{(q_0, c = 0), (q_1, c = 0)\}$; and $(\{q_0, q_1\}, c \in \{0, 1\})^{DFA} = \{(q_0, c = 0), (q_1, c = 0), (q_1, c = 1)\}$ (recall that q_0 is not in the scope of c, hence c has implicitly the value 0 there). □

6.4 Uniformity: A Sufficient Semantic Correctness Criterion

Given a CA A, we produce a CsA A' that may overapproximate A in terms of the language. We explain how this may happen and present conditions under which the language stays unchanged. In particular, the overapproximation is caused by non-Cartesian powerstates of $DFA(A)$. (Recall that, in a Cartesian powerstate, states in the scope of a counter must appear with the same set of values of that counter.) A configuration of the CsA cannot encode a non-Cartesian powerstate precisely, it can only overapproximate it. A larger powerstate may then accept a larger language.

Example 6.5. Take $R =$ `(a|aa){5}` and the CA(R) shown in Fig. 4. After reading the word *aa*, *DFA*(CA(R)) reaches the powerstate $\{(q_0, c = 1), (q_0, c = 2), (q_1, c = 2)\}$, which is not Cartesian because both states are in the scope of the counter c but are paired with different counter values. Our CsA would reach the configuration $(\{q_0, q_1\}, c \in \{0, 1, 2\})$, which encodes the larger powerstate $\{(q_0, c = 0), (q_0, c = 1), (q_0, c = 2), (q_1, c = 0), (q_1, c = 1), (q_1, c = 2)\}$ where both states appear with both counter values. □

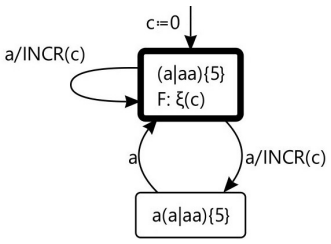


Fig. 4. $CA($`(a|aa){5}`$)$.

Uniformity. We now introduce the so-called *uniformity* of a CA as a property under which determinization preserves the language. Uniformity prevents creation of non-Cartesian powerstates. It includes two conditions.

The first condition prevents the kind of scenario from Example 6.5. For each DFA transition τ', it requires that every CA state q that is in the scope of some counter c within the DFA state to which τ' leads receives the same set of values of c. This requires testing whether the sets of transitions covered by τ' and incoming to every such CA state q induce the same CsA operations for c.

The second condition prohibits two counters from being active at once, a scenario which arises from regexes with nested counting. Indeed, the relation between values of two simultaneously active counters may easily become more intricate than what can be expressed by a Cartesian product of two sets (consider, e.g., the regex $a?(a\{1\}a)\{2\}$ and the word *aaa*). The condition requires testing that no state appears in the scope of two counters.

Formally, given a CsA transition $\tau' = S \text{-}(\alpha \wedge \beta, f')\!\!\rightarrow T$, a counter c, and a CA state $q \in \sigma(c)$, we define the set $f'_q(c)$ of *incoming CsA operations* for c induced by the incoming transitions of q from which τ' is built (α-transitions consistent with β originating in S) as follows:

$$f'_q(c) \stackrel{\text{def}}{=} \{op(\tau, c) \mid \tau \in \Delta_{S,\alpha,\beta(c)} \wedge \text{the target of } \tau \text{ is } q\} \ .$$

We call the transition τ' *uniform* iff, for each counter $c \in C$, any two states $q, r \in \sigma(c) \cap T$ have the same sets of incoming CsA operations, i.e., $f'_q(c) = f'_r(c)$. The CA A is then *uniform* if all transitions of A' are uniform and if no state of A appears in the scope of two counters.

Theorem 6.6. *If a CA A is uniform, then $\mathcal{L}(A) = \mathcal{L}(A')$.*

Proof (idea). By showing bisimilarity between states q of *FA*(A'), i.e., configurations of the CsA A' and powerstates q^{DFA} of *DFA*(A). □

Uniformity can be checked on the fly, while constructing A'. It is also automatically implied when the CA is constructed from certain classes of regexes, as discussed below.

6.5 Syntactic Correctness Criteria

Uniformity is only a semantic property. Below, we show examples of actual regexes that do and do not lead to uniform CAs and discuss some simple syntactic classes of regexes that imply uniformity. A detailed study of syntactic classes of regexes that guarantee uniformity is, however, beyond the scope of this paper and a part of our future work.

The regexes that induce non-uniform CAs are often those where, intuitively, there is a position in some input text that may either be matched against the first character of a counted sub-expression or against some inner character of the same sub-expression. In such a situation, there may be two runs of the induced CA: one that increments the associated counter (the increment happens) at that position and moves to some state q, and the other that leaves the counter as it is, while in its scope, and moves into a different state r. The counter value then depends on the state: it is different in q and in r. The corresponding DFA state is then non-Cartesian and the CA is non-uniform.

Example 6.7. We present several commented examples of regexes with non-uniform CAs where our determinization overapproximates the language of the obtained CsA.

- `(a|ab|ba){5}` — the string *aba* could be matched as "a" followed by "ba", having incremented the counter twice, or as "ab" that is followed by the prefix "a" of "ab", having incremented the counter once only.
- `a{1,3}a{3}` — this case can be explained similarly as the previous one. Alternatively, note that, assuming that our translation to a CA produces two counters, say c_1 and c_2, then after reading n letters a, the CA needs to remember that $c_1 + c_2 = n$. Such non-trivial relations between counter values are not Cartesian.
- `.*(aa){6}` — assuming a sequence of a's on the input, the counter may be either incremented on odd characters and left unchanged on even ones, or the other way around. As the counter values depend on the position within the "aa" (and hence on the CA state), the CA cannot be uniform. Note that the prefix `.*` is quite usual as it corresponds to searching for the regex `(aa){6}` anywhere in the input string.
- `.*(a{2}){2}` — after reading *aa*, if the value of the outer counter is 1, then the value of the inner counter must be 0. This is a non-trivial relation between the values of the two counters, which is not Cartesian. Nested counting is often problematic, however, many of such examples may still be solved quite efficiently by unfolding one of the counters. □

Syntactic classes of regexes that guarantee uniformity. A simple class of regexes that guarantees uniformity is a generalization of the class of *monadic* regexes of [Holík et al. 2019] (where counting is allowed over character classes only). Namely, it consists of regexes with counting loops of the form

$$(\alpha_1 \dots \alpha_n)\{\ell, k\} \text{ s.t. } [\![\alpha_1]\!] \text{ is disjoint from every } [\![\alpha_i]\!], 1 < i \leq n.$$

Intuitively, the disjointness with α_1 ensures that the generated CA will only be able to process α_1 through an increment transition at the beginning of a new iteration of the loop, with no possibility of having a conflicting NOOP transition that could read the same symbol inside the body of the loop (which is exactly what happens with the second symbol a in Example 6.5). The CsA compiled form this class are also guaranteed to be fast.

7 EXPERIMENTAL EVALUATION

We have implemented our approach in a C# prototype called `CA` available at [Turoňová et al. [n.d.]] (see [Turoňová et al. 2020] for details how to efficiently implement CsAs) and evaluated its pattern matching capabilities against other state-of-the-art regex matchers on patterns that use the counting operator. We focused on comparison against Google's `RE2` library [Google [n.d.]][7], an automata-based matcher designed to be fast, predictable, and resilient against ReDoS attacks. We also include other three efficient matchers into the comparison, namely the standard GNU `grep` program [Haertel et al. [n.d.]] (version 3.3), the `.NET` standard library regex matcher from `System.Text.RegularExpressions` [Microsoft 2020], and Symbolic Regex Matcher (`SRM`) [Saarikivi et al. 2019].

Let us shortly summarize how the tools work. The main algorithms of `RE2` and `grep` implement optimized versions of the Thompson's on-the-fly determinization where the constructed DFA states are cached. The construction has a bound on the size of the DFA—if the bound is reached, the so-far constructed DFA states are flushed to avoid consuming too much memory. In some situations when caching is found ineffectual, `RE2` turns the caching off, and the performance can drop even lower (see the description in [Cox 2010] for details). We note that `RE2` rejects an input regex if it contains a counting operator with a bound bigger than 1,000. `SRM` is based on *symbolic derivatives*

[7] We used the version 2019-01-01 of `RE2` via the command line interface `re2g` from https://github.com/akamai/re2g.

constructed on the fly, also in the spirit of the Thompson's algorithm, and, likewise, bases its efficiency on caching (in fact, `SRM` is quite close to an implementation of the Thompson's algorithm over CAs with caching). The `.NET` matcher uses a backtracking algorithm over NFAs, while our `CA` eagerly constructs a deterministic CsA for the input regex. The former four are mature tools, and especially `RE2` and `grep` contain many high- and low-level optimizations, such as using the Boyer-Moore algorithm [Boyer and Moore 1977] to skip over many characters that are known to not be a part of a match. `RE2` and `grep` are compiled programs while `CA`, `SRM`, and `.NET` run within the .NET Framework (therefore, they have some inherent overhead due to the *just-in-time* compilation at start-up and its inability to use advanced code optimizations, as well as garbage collection). Note that even though the tools based on the on-the-fly subset construction (`RE2`, `grep`, and `SRM`) are linear to the lenght of the text, they still take space exponential to the counter bounds in the worst case, by creating sets of the size linear to the counter bounds, exponential to their decadic encoding used in the regex.

We run our benchmarks on a machine with the Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40 GHz running Debian GNU/Linux (we use the Mono platform [project [n.d.]] to run .NET tools). To avoid issues with generating exact matches, which might differ for different tools, the tools were run in the setting where they counted the number of lines matching[8] the given regex (e.g. the `-c` flag of `grep`).

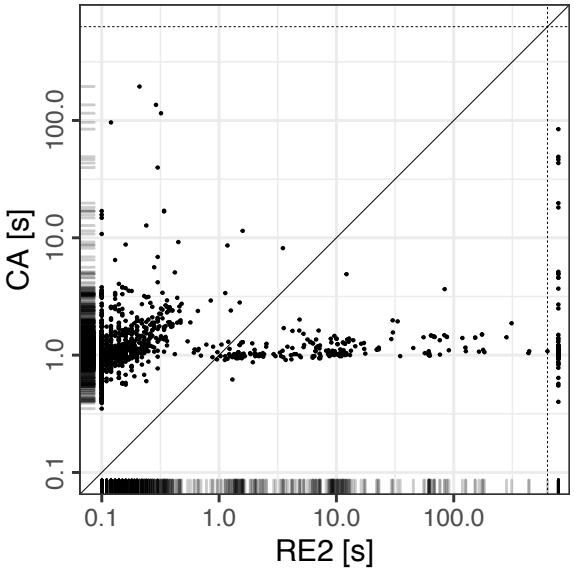
7.1 ReDoS Resiliency

Our main experiment focuses on the resilience of the matching engines against ReDoS attacks. The regexes used for this experiment were selected (1) from the database of over 500,000 real-world regexes coming from an Internet-wide analysis of regexes collected from over 190,000 software projects [Davis et al. 2019]; (2) from databases of regexes used by *network intrusion detection systems* (NIDSes), in particular, Snort [M. Roesch et al. [n.d.]], Bro [Robin Sommer et al. [n.d.]], Sagan [The Sagan team [n.d.]], and the academic papers [Češka et al. 2018; Yang et al. 2010]; (4) the RegExLib database of regexes [RegExLib.com [n.d.]]; and (5) industrial regexes from [Holík et al. 2019], used for security purposes. From these, we created our set of benchmarks by the following steps:

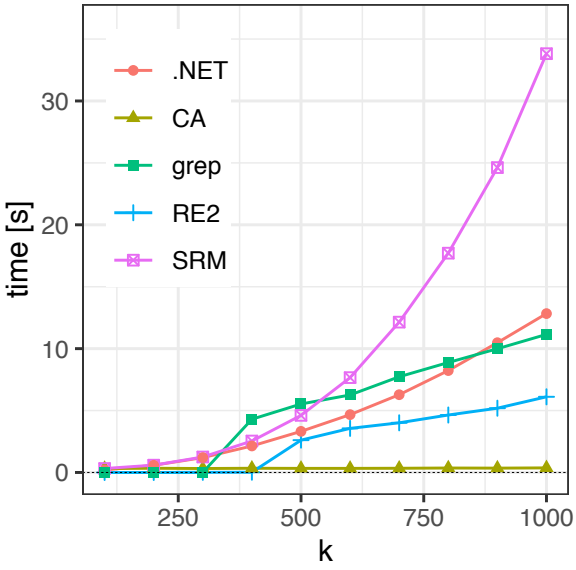
(1) We selected regexes that contained counting loops whose sum of upper bounds was larger than 20. This let us focus on regexes where the use of counting makes sense (there are surprisingly many regexes occurring in practice where the use of a counting loop is unnecessary, e.g., regexes containing sub-expressions similar to `a{0,1}` or even just `a{1}`). Moreover, we also removed all except 26 regexes with counters bigger than 1,000, which cannot be handled by `RE2`. We left the 26 regexes as representatives of "large" counters. This left us with 5,000 regexes.
(2) Then, we filtered out regexes R such that either CA(R) was not uniform (i.e., the CsA produced by our algorithm was not precise, cf. Section 6.4), or such that the *CsA* was not fast (i.e. not all counting-set operators were constant-time, cf. Section 6.1). After this step, a vast majority, 4,429 of the regexes, remained.
(3) For the regexes that remained, we used a lightweight ReDoS generator designed to exploit counting (cf. Section 7.3) to generate ~10 MiB long input texts. In particular, we managed to generate "adversarial" input texts for 1,789 regexes (for the rest of the regexes, either the underlying state space was too small, so the generator could not construct the text, or the generation hit the timeout of 600 s). Our benchmark data set is available at [Holík et al. 2020].

We ran all tools on the generated benchmarks (counting the number of lines of the input text matching the regex) and give scatter plots comparing the running times of the tools in Fig. 5a

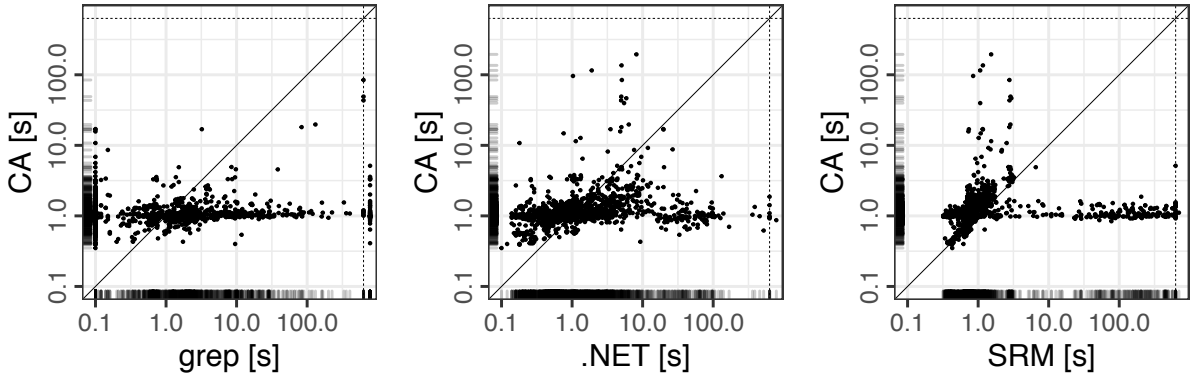
[8] We consider the standard semantics of "matching" used by `grep`, i.e., a line matches a regex R if it contains a string that is in $\mathcal{L}(R)$, unless it contains start-of-line (`^`) or end-of-line (`$`) anchors, in which case the matched string needs to occur at the start and/or at the end of the line respectively.



(a) The comparison of running times of CA and RE2 on our benchmark set (CA wins: 287/1,789)



(b) Running times of the tools on the regex "(_a)$\{k\}$_a" where k is a parameter



(CA wins: 862/1,425) (CA wins: 708/1,789) (CA wins: 345/1,789)

(c) The comparison of running times of CA with grep, .NET, and SRM on our benchmark set

Fig. 5. Graphs with results of our experiments. Note that, in (a) and (c), the axes are logarithmic, the dashed lines denote the timeout (600 s), and the data points between the dashed lines and the edge of a plot represent benchmarks where the tool did not run successfully. We also provide the number of times CA won.

and Fig. 5c (the timeout was 600 s). On the bottom and the left-hand side of every plot, there are rug plots illustrating the distribution of the data points. Note that the axes are logarithmic, so the difference between data points grows as these points are away from zero (in particular, differences of values smaller than 1 s are negligible). The semantics of regexes supported by grep differs from the one supported by other tools, so we only considered the cases when the number of matches was the same when comparing with grep). In the plots, the data points between the dashed lines and edges of the plots represent errors, e.g. due to the regex being rejected (for counters >1,000 for RE2) or being interpreted using a different semantics (in the case of grep).

In Fig. 5a, we compare CA with RE2. We wish to point out the following interesting observations. Although RE2 wins more often on the whole benchmark set (our prototype does not include the many advanced optimizations present in RE2), there is a number of benchmarks (287) where its performance significantly deteriorates, and CA is faster. In particular, there are 89 benchmarks

Table 1. Statistics for the graphs in Fig. 5 (times are given in seconds). For CA, we provide several times: "total" is the total time, "CA" is the time for translating a regex into a (nondeterministic) CA, "CsA" is the time of determinization of the CA into a CsA, and "match" is the time spent when matching the input text.

	RE2	grep	.NET	SRM	CA			
					total	CA	CsA	match
mean	36.11	34.38	9.12	26.78	1.73	0.05	0.23	0.69
median	0.10	0.70	0.76	0.73	1.03	0.03	0.04	0.68
std. dev	157.05	147.17	52.10	106.16	7.27	0.29	2.73	0.29
timeouts	1	11	8	16	0			

where the time of RE2 is bigger than 10 s, i.e., its speed drops below 1 MiB/s (we consider this speed of processing denotes a successful ReDoS attack, even though the limit may be significantly larger in practice[9]). For CA, the number of benchmarks that took over 10 s was only 22; in fact, all except 3 benchmarks finished within 100 s—the blow-up in these 3 benchmarks is not caused by the counters but rather by many "|" and "?" operators, so over 70 % of the total time is spent by constructing the CsA. If used, e.g., in an NIDS, the CsA would be created only once and then used for matching giga-/terabytes of data, so the initial overhead could be neglected.

Comparing with the other tools (Fig. 5c) and also clearly visible in the corresponding rug plots and the statistics in Table 1, we can observe that the performance of CA is much more robust than the performance of the other tools; the mean time and standard deviation of CA is significantly lower than the rest of the tools. In particular, from the benchmarks where CA was faster than RE2, the time of CA on all except two benchmarks was almost the same (including them, the standard deviation was 0.37). We provide four times for CA: "total": the total user time of matching (measured using the GNU `time` utility), "CA": the time for translating the input regex into a CA, "CsA": the time it took to determinize the CA into a CsA, and "match": the time of matching the input text with the CsA. Note that, in the tables, there is a noticeable discrepancy between the sum "CA" + "CsA" + "match" and "total", which is due to the .NET Framework overhead, such as just-in-time compilation and (in particular) the garbage collector.

In Table 2, we give a selection of interesting benchmarks. These contain benchmarks that are difficult for usually more than one tool. We emphasize the benchmarks coming from the NIDSes Snort and Bro. Notice that, for most of them, matching using RE2 (and also other tools) gets extremely slow. Slow matching over these regexes can have disastrous consequences for network security, potentially completely eliminating a given NIDS.

The CsAs produced by CA were also much smaller than the corresponding DFAs. The CsAs have on average 29 states (median: 7) and 306 transitions (median: 11). On the other hand, classical NFAs constructed from the regexes have on average 112 states (median: 52), and when determinized, the resulting DFAs have on average 2,802 states (median: 67) and 10,384 transitions (median: 107). Using CsAs significantly lowers the chance that determinization explodes.

7.1.1 The Effect of Nondeterministic Counting. We say that a regex contains *nondeterministic counting* if, when translated into a CA A using the algorithm in Section 5, there is a word w such that A can over w reach two configurations with different values of some counter.

Regexes with nondeterministic counting are the main focus of our benchmark. Namely, they constitute 67 % of the 1,789 regexes used. From the 1,284 regexes that were at least *slightly* problematic for some of the other tools except CA (it took some tool ≥ 1 s), 73 % of them were with

[9]The required processing speed depends on the application. NIDSes performing deep packet inspection may require a line-processing speed of units or tens of GiB/s [Češka et al. 2018], while application servers validating user inputs may suffice with units or tens of MiB/s.

Table 2. Selection of interesting benchmarks. "TO" denotes a timeout (600 s) and "—" denotes an error. Due to space constraints, in the "Regex" column, "…" denotes omitted parts of the regexes (we tried to preserve the parts containing occurrences of the repetition operator) and "~" denotes breaking a regex into two lines. In the column source, Sw denotes the regexes collected in [Davis et al. 2019] from software projects.

Source	Regex	RE2	grep	.NET	SRM	CA			
						total	CA	CsA	match
Snort	`.*[aA][uU][tT][hH]...[iI][cC] ~ ~[^\x0A]{512}`	11.27	7.8	361.1	555.56	1.04	0.03	0.05	0.31
Snort	`\x20[^\x21\x22]{500}`	439.98	0.11	2.20	TO	1.08	0.03	0.04	0.83
Snort	`^RCPT TO\x20\s*[\w\s@\.]{200,}~ ~\x20[\w\s@\.]{200,}...`	340.7	—	TO	TO	1.68	0.03	0.07	0.89
Snort	`php.*\x20[^\n]{256}`	176.75	0.10	1.22	TO	1.08	0.04	0.07	0.74
Snort	`^(NT|CallBack|SID|TimeOut)\s*~ ~\x20\s*[^\n]{512}`	164.11	0.12	14.59	229.41	1.07	0.03	0.07	0.72
Snort	`.*[nN][eE][wW]... [^\x20]{100}`	0.13	1.26	39.92	0.74	0.81	0.03	0.04	0.65
Bro	`^[nN][aA][mM][eE]=s*[^\r\n\x3b~ ~\x20\x09\x0b\x2c]{300}`	128.57	12.24	0.51	76.48	1.15	0.03	0.04	0.94
Sw	`_.{39}`	22.96	225.34	1.94	357.68	1.12	0.03	0.04	0.79
Sw	`(.{1,980}[,])\s+(\S)`	260.59	TO	308.66	0.63	1.07	0.03	0.05	0.59
Sw	`(_a ){64999}_a`	—	—	TO	TO	0.96	0.03	0.04	0.51
Sw	`\[{50000}a\]{50000}`	—	—	4.36	TO	5.13	0.02	0.02	0.41
Sw	`^QS([NDR])(.{4})(.{6})(\d{8})...~ ~(.{4})(.{6})(.{8})(.{8})(.)$`	0.12	0.10	1.03	0.85	96.20	0.04	81.64	0.65

nondeterministic counting. From the 454 regexes that were *significantly* problematic for some of the other tools (it took some tool ≥ 10 s), 85 % of them had nondeterministic counting. From the 109 regexes that were *problematic* for *all* other tools (≥ 1 s), 100 % were with nondeterministic counting. As shown in the results above, our approach can deal with nondeterministic counting quite well.

7.1.2 Adversarial Regexes. Another ReDoS scenario is when the attacker can control the regex to be used for matching. Creating a counting regex causing efficiency problems for a given text is easier than generating adversarial texts. For instance, the regex `[a-zA-Z().,' ]*[a-zA-Z ][a-zA-Z();' ]{250}` was obtained as a modification of the running example "`.*a.`{k}" (where `a` appears k positions from the end). When run on a ~4 MiB English text with sufficiently long lines, RE2 took 86 s, grep took 26 s, while CA took only 1.1 s. Similar examples could be obtained from regexes from Section 7.1 for which some specific difficult text can be generated, namely by widening their character classes.. Our approach solves a large class of the dangerous cases, allowing one to significantly alleviate restrictions put on the user for security/efficiency reasons.

7.2 Robustness wrt Counter Values

This experiment measures the ability of the tools to cope with increasing counter bounds. For this, we selected the regex "`(_a ){`k`}_a`" where k is a parameter (the original regex `(_a ){64999}_a` comes from [Davis et al. 2019]) and measured the time the tools took on a ~500 KiB text created by our generator for increasing values of k. We give the results in Fig. 5b (the timeout was 40 s).

With the increasing value of k, the time needed by CA stays constant, around 0.35 s, while the time needed by other tools grows. In particular, .NET and SRM have cubic trends wrt the value of k, while RE2 and grep grow linearly. Notice that, for RE2 and grep, their matching time is low (around 0.01 s) until they reach a threshold from which they start behaving linearly. This corresponds to the situation when the size of the cache for storing states of the NFA-to-DFA construction is not enough to accommodate the DFA states exercised by the input adversarial text. This yields repeated flushing of the cache, making it ineffectual.

7.3 Adversarial Text Generation

RE2 and grep store powerstates of the NFA-to-DFA construction in a cache. In typical cases, the amount of cache misses is low and almost the entire text is processed using the cache, which is extremely fast. If the cache, however, exceeds a given size, it is flushed. If the input text is such that the DFA run sees many different states, then cache misses are frequent, so large powerstates need to be constructed often, and the performance of the matching drops.

Therefore, we focus on generating texts that force exploration of many new large powerstates. In essence, we explore the configuration space of the CsA with the goal of finding as many large configurations as possible, with the focus on generating large counting sets. We partially drive the search towards loops in the CsA structure that have a potential to create large counting sets: the loops use counters with large bounds, do not contain exits, and contain Rst or Rst1 operations. For space reasons, we omit the technical details here; perfecting this method for stress testing automata-based matchers is, however, one of our future goals.

7.4 A Note on the Maturity of the Tools

The aim of our experiments is comparing algorithms rather than tools, and it should be noted that CA is much less optimized than the rest. This holds especially for RE2 and grep, which have both been actively developed for over 10 years and the amount of engineering effort invested into making them fast is substantial. The optimizations are both high-level, such as using the Boyer-Moore algorithm for skipping sections of the input text, and low-level, such as using C/C++, on-the-fly determinization, or optimizing memory accesses [Cox 2010; Haertel [n.d.]]. On the other hand, although there have been some optimizations done in CA (such as finding a start of a match), their nature is still quite simple. The three tools are, however, all based on the same principle of using deterministic automata, and many of the optimizations and heuristics in RE2 and grep (at least all of those mentioned above) could be directly re-applied in our setting. SRM builds on the .NET framework and reuses the .NET regex parser while replacing the built-in backtracking back-end matcher with a matching engine based on Brzozowski-style symbolic derivatives to create the DFA on the fly. In fact, CA builds on the open-source codebase of SRM and extends it with counters.

8 RELATED WORK

Regexes and their derivatives. Brzozowski derivatives [Brzozowski 1964] provide a practical approach to incrementally creating a DFA from a regex and can be used for efficient matching [Fischer et al. 2010; Owens et al. 2009] and match generation [Saarikivi et al. 2019]. Efficient determinization based on Brzozowski derivatives was first investigated in [Berry and Sethi 1986]. In the classical setting, Antimirov derivatives [Antimirov 1996] are used to construct NFAs from regexes, and may in some cases result in exponentially more succinct automata than the corresponding DFAs constructed with Brzozowski derivatives. The precise connection between conditional derivatives defined in Section 5 and Antimirov derivatives is that, without counting loops, $\{D \mid \langle \textsc{id}, D\rangle \in \partial_a(S)\}$ is exactly the Antimirov derivative of R for a. The Antimirov construction has also been generalized to extended regexes [Caron et al. 2011] allowing Boolean operators such as complement and intersection. Basic theoretical properties between various automata formalisms and derivatives are discussed in [Allauzen and Mohri 2006].

Automata with counting. This work is a continuation of our recent work [Holík et al. 2019]. In [Holík et al. 2019], we propose a general determinization of CAs that can produce smaller automata than the naive explicit determinization but has the same worst-case complexity, which depends on the counters with the factor $(K+1)^{|C|}$ where C is the set of counters and K the maximum counter upper bound. It also proposes a more efficient algorithm for the class of monadic regexes

(single-state-scoped counters and counting on self-loops only), but it can still generate $(K+1)^{|Q|}$ states (e.g., it would generate $K+1$ states for the regex from Fig. 1)—while the complexity of our determinization does not depend on K. The work [Holík et al. 2019] does also not present a derivative construction for translating regexes into CAs nor an application of CAs in pattern matching.

The use of counters has also been investigated in [Björklund et al. 2015] for regexes with bounded repetition, building on the formalism of counter automata called CNFAs [Gelade et al. 2012]. A CA in the current paper is essentially a symbolic generalization of a CNFA with some small technical differences, such as counters being 0-based as opposed to 1-based in a CNFA. The latter difference is mainly due to our use of a generalized *Antimirov* construction of CAs, as opposed to a generalized *Glushkov* construction used in [Gelade et al. 2012], which is algorithmically quite different. The work in [Björklund et al. 2015] focuses mostly on deterministic regexes and on a different problem, namely, the so-called incremental matching in the context of database queries (a query is repeatedly evaluated on a gradually changing word). For standard matching, it uses a variant of the Thompson's algorithm applied directly on a CA instead of an NFA (hence the translation of the regex to an automaton does not depend on the counter bounds, but each text character is processed with the same cost as with the original Thompson's algorithm, at worst linear to the size of the NFA and the counter bounds). This algorithm is indeed fast on deterministic regexes from practice but can slow down significantly on nondeterministic ones (which we witnessed in several experiments with the prototype implementation of [Björklund et al. 2015] on several of our regexes).

The work in [Kilpeläinen and Tuhkanen 2003] is a theoretical study of matching regexes with counting. It proposes a matching algorithm based on dynamic programming that runs in time at worst quadratic to the length of the text (while determinization and NFA-simulation-based algorithms run in time linear to the text length). The experimental comparison of [Björklund et al. 2015] with their variant of Thompson's algorithm suggests that the matching algorithm of [Kilpeläinen and Tuhkanen 2003] is indeed not competitive in practice.

Extended FAs (XFAs) augment classical automata with a scratch memory of bits [Smith et al. 2008a,b] that can represent counters. Regexes are compiled into deterministic XFAs by first using an extended version of the Thompson's algorithm, followed by an extended version of the classical powerset construction and minimization. Although a small XFA may exist, the determinization algorithm incurs an intermediate exponential blowup of the search space for inputs such as `.*a.{`k`}`.

R-automata [Abdulla et al. 2008] are also related to our CAs, but their counters need not have upper bounds and cannot be tested or compared. Further, there are various notions of extended finite state machines whose expressive power goes beyond regular languages, e.g., [Bardin et al. 2008; Cheng and Krishnakumar 1993; Shiple et al. 1998; Smith et al. 2008b]. Such automata are, however, not suitable for the problem of pattern matching considered here.

Regexes with counting. Regexes with counters are also discussed in [Gelade et al. 2007; Hovland 2009; Kilpeläinen and Tuhkanen 2007]. The automata with counters used in [Hovland 2009], called FACs, are close to our CAs, but we allow symbolic character predicates and more kinds of counter updates. The conversion from regexes to FACs proposed in [Hovland 2009] uses a variant of Glushkov automata [Glushkov 1961] and the `first-last-follow` construction [Berstel and Pin 1996; Brüggemann-Klein and Wood 1998]. For us, the Antimirov-derivative-based construction was easier to implement and provides benefits that are not available otherwise. Namely, it allows subsumption checking between regexes, and it generates fewer counters (one per distinct counter sub-expression rather than one per counter position in the regex abstract syntax tree). While all these algorithms generate ϵ-free automata, they differ in complexity [Allauzen and Mohri 2006] and are thus not merely different disguises of the same technique. In particular, the Antimirov automaton is in general smaller than the Glushkov automaton with up to $n+1$ states and up to n^2 transitions.

The Antimirov automaton is in fact a quotient of the Glushkov automaton [Champarnaud and Ziadi 2001; Ilie and Yu 2003]. Another generalization of Antimirov derivatives [Lombardy and Sakarovitch 2005] introduces expressions kR where R is a rational expression and k a multiplicity from a semiring such as $\mathbb{Q}$; this generalization is unrelated to counters.

An open question is whether the generalized Antimirov construction can be extended to work with Brzozowski derivatives [Brzozowski 1964]; we believe that such an extension, if it exists, is not straightforward because it would give rise to a direct and incremental determinization algorithm.

There are also works on regexes with counting that translate deterministic regexes to CAs and work with different notions of determinism [Chen and Lu 2015; Gelade et al. 2012]. A central result in [Hovland 2009] is that *counter-1-unambiguous* regexes can be compiled into deterministic FACs and that checking determinism of FACs can be done in polynomial time. The related work in [Hovland 2012] studies membership in regexes with counting. None of these papers addresses the problem of determinizing nondeterministic CAs.

Pattern matching of regexes with counting. The counting operator often appears in regexes in practice. In particular, our analysis of the 537k real-world regexes obtained in the study performed by Davis et al. [Davis et al. 2019] showed that over 33k regexes contained the counting operator.

GNU `grep` [Haertel et al. [n.d.]] (written in C) and `RE2` [Google [n.d.]] (written in C++) are extremely optimized regex matchers. Both are based on translating the regex into an NFA and performing an on-the-fly determinization during the matching, avoiding a costly *a priori* determinization, while keeping a good performance by avoiding backtracking. (The translation into FAs is only allowed when the regex does not include back-references, which allow one to express some context-free properties). Both engines process the counting operator by first rewriting a regex of the form `<re>{`n,m`}` into `<re>. . . <re><re>{0,`$m-n$`}`. The regex `<re>{0,`k`}` is then transformed into `(<re>(<re>(. . . <re>?)?)?)?` (see [Cox 2010] for more details).

In the .NET ecosystem, we are aware of two regex matchers. The first one is the standard .NET regex matcher provided in `System.Text.RegularExpressions`, which is based on a backtracking search. The other one is Symbolic Regex Matcher (SRM) of [Saarikivi et al. 2019] based on the so-called *symbolic derivatives*, which provide a backtracking-free search (without an explicit conversion into a DFA) and can deal more efficiently with the counting operator.

9 CONCLUSIONS AND FUTURE WORK

We have presented a framework for efficient pattern matching of regexes with counting, which includes a derivative construction to compile regexes to counting automata, their subsequent determinization into novel counting-set automata, and a fast matching algorithm. The resources needed to build the CsAs are independent of counter bounds. It handles a majority of regexes with counting found in practice, with a much more stable performance than other matchers.

In the future, we intend to explore the limits of the idea of counting sets to enlarge and clearly delimit the class of regexes and counting automata that can be succinctly determinized while preserving fast matching. We also plan to explore possible usage of CsAs as a replacement of classical automata in other applications where automata are used, for instance, as symbolic representations of state spaces. For this, we intend to develop CsA counterparts of essential automata techniques, such as Boolean operations and minimization/size-reduction techniques. We also wish to elaborate on our method for generating texts for stress-testing matchers on regexes with counting.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and also Juraj Síč for their valuable comments and suggestions. This work is supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, and the FIT BUT internal project FIT-S-20-6427.

REFERENCES

Parosh Aziz Abdulla, Pavel Krčál, and Wang Yi. 2008. R-Automata. In *CONCUR'08 (LNCS, Vol. 5201)*. Springer, 67–81. https://doi.org/10.1007/978-3-540-85361-9_9

Cyril Allauzen and Mehryar Mohri. 2006. A Unified Construction of the Glushkov, Follow, and Antimirov Automata. In *Mathematical Foundations of Computer Science 2006*. Springer Berlin Heidelberg, Berlin, Heidelberg, 110–121. https://doi.org/10.1007/11821069_10

Valentin Antimirov. 1996. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science* 155, 2 (1996), 291 – 319. https://doi.org/10.1016/0304-3975(95)00182-4

Adam Baldwin. 2016. *Regular Expression Denial of Service affecting Express.js*. http://web.archive.org/web/20170116160113/https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43

Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. 2008. FAST: acceleration from theory to practice. *STTT* 10, 5 (2008), 401–424. https://doi.org/10.1007/s10009-008-0064-3

Gerard Berry and Ravi Sethi. 1986. From regular expressions to deterministic automata. *Theoretical Computer Science* 48, 3 (1986), 117–126. https://doi.org/10.1016/0304-3975(86)90088-5

Jean Berstel and Jean-Éric Pin. 1996. Local languages and the Berry-Sethi algorithm. *Theoret. Comput. Sci.* 155, 2 (1996), 439–446. https://doi.org/10.1016/0304-3975(95)00104-2

Henrik Björklund, Wim Martens, and Thomas Timm. 2015. Efficient Incremental Evaluation of Succinct Regular Expressions. In *CIKM'15 (ACM)*. https://doi.org/10.1145/2806416.2806434

Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (Oct. 1977), 762–772. https://doi.org/10.1145/359842.359859

Anne Brüggemann-Klein and Derick Wood. 1998. One-unambiguous regular languages. *Information and Computation* 140, 2 (1998), 229–253. https://doi.org/10.1006/inco.1997.2695

Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. 2011. Partial Derivatives of an Extended Regular Expression. In *Language and Automata Theory and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 179–191. https://doi.org/10.1007/978-3-642-21254-3_13

Jean-Marc Champarnaud and Djelloul Ziadi. 2001. Computing the equation automaton of a regular expression in $O(s^2)$ space and time. In *Proceedings of CPM 2001 (LNCS, Vol. 2089)*. Springer, 157–168. https://doi.org/10.1007/3-540-48194-X_15

Haiming Chen and Ping Lu. 2015. Checking determinism of regular expressions with counting. *Information and Computation* 241 (2015), 302 – 320. https://doi.org/10.1016/j.ic.2014.12.001

Kwang-Ting Cheng and A. S. Krishnakumar. 1993. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*. ACM Press, 86–91. https://doi.org/10.1145/157485.164585

Wikipedia contributors. 2019. *Regular expression—Wikipedia*. https://en.wikipedia.org/w/index.php?title=Regular_expression&%20oldid=852858998

Russ Cox. 2010. Regular Expression Matching in the Wild. https://swtch.com/~rsc/regexp/regexp3.html.

Loris D'Antoni and Margus Veanes. 2020. Automata Modulo Theories. *Commun. ACM* (2020).

James C. Davis. 2019. Rethinking Regex Engines to Address ReDoS. In *Proceedings of ESEC/FSE'19* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 1256–1258. https://doi.org/10.1145/3338906.3342509

James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of ESEC/FSE'18* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/3236024.3236027

James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-use and Portability of Regular Expressions. In *Proceedings of ESEC/FSE'19* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 1256–1258. https://doi.org/10.1145/3338906.3338909

Stack Exchange. 2016. *Outage Postmortem*. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016

Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. A Play on Regular Expressions: Functional Pearl. *SIGPLAN Not.* 45, 9 (2010), 357–368. https://doi.org/10.1145/1863543.1863594

Wouter Gelade, Marc Gyssens, and Wim Martens. 2012. Regular Expressions with Counting: Weak versus Strong Determinism. *SIAM J. Comput.* 41, 1 (2012), 160–190. https://doi.org/10.1137/100814196 Extended version of paper in MFCS'09.

Wouter Gelade, Wim Martens, and Frank Neven. 2007. Optimizing schema languages for XML: Numerical constraints and interleaving. In *Proceedings of ICDT'07 (LNCS, Vol. 4353)*. Springer, 269–283. https://doi.org/10.1007/11965893_19

V. M. Glushkov. 1961. The abstract theory of automata. *Russian Math. Surveys* 16 (1961), 1–53. https://doi.org/10.1070/RM1961v016n05ABEH004112

Google. [n.d.]. RE2. https://github.com/google/re2.
John Graham-Cumming. 2019. *Details of the Cloudflare outage on July 2, 2019.* https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/
Mike Haertel. [n.d.]. why GNU grep is fast. https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html.
Mike Haertel et al. [n.d.]. GNU grep. https://www.gnu.org/software/grep/.
Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. 2019. Succinct Determinisation of Counting Automata via Sphere Construction. In *Proc. of APLAS'19 (LNCS, Vol. 11893)*. Springer, 468–489. https://doi.org/10.1007/978-3-030-34175-6_24
Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. 2020. *Dataset for the OOPSLA'20 paper "Regex Matching with Counting-Set Automata".* https://doi.org/10.5281/zenodo.3974360
Dag Hovland. 2009. Regular Expressions with Numerical Constraints and Automata with Counters. In *ICTAC (LNCS, Vol. 5684)*. Springer, 231–245. https://doi.org/10.1007/978-3-642-03466-4_15
Dag Hovland. 2012. The Membership Problem for Regular Expressions with Unordered Concatenation and Numerical Constraints. In *Language and Automata Theory and Applications*. Springer Berlin Heidelberg, Berlin, Heidelberg, 313–324. https://doi.org/10.1007/978-3-642-28332-1_27
Lucian Ilie and Sheng Yu. 2003. Follow automata. *Information and Computation* 186, 1 (2003), 146–162. https://doi.org/10.1016/S0890-5401(03)00090-7
Pekka Kilpeläinen and Rauno Tuhkanen. 2003. Regular Expressions with Numerical Occurrence Indicators - preliminary results. In *Proceedings of the Eighth Symposium on Programming Languages and Software Tools, SPLST'03, Kuopio, Finland, June 17-18, 2003*. University of Kuopio, Department of Computer Science, 163–173.
Pekka Kilpeläinen and Rauno Tuhkanen. 2007. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation* 205, 6 (2007), 890–916. https://doi.org/10.1016/j.ic.2006.12.003
Sylvain Lombardy and Jacques Sakarovitch. 2005. Derivatives of rational expressions with multiplicity. *Theoretical Computer Science* 332, 1 (2005), 141 – 177. https://doi.org/10.1016/j.tcs.2004.10.016
M. Roesch et al. [n.d.]. Snort: A Network Intrusion Detection and Prevention System,. http://www.snort.org.
Microsoft. 2020. . https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex.match
Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression Derivatives Re-examined. *J. Funct. Program.* 19, 2 (2009), 173–190. https://doi.org/10.1017/S0956796808007090
Mono project. [n.d.]. Mono. https://www.mono-project.com/.
RegExLib.com. [n.d.]. The Internet's first Regular Expression Library, . http://regexlib.com/.
Robin Sommer et al. [n.d.]. The Bro Network Security Monitor. http://www.bro.org.
Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. 2019. Symbolic Regex Matcher. In *TACAS'2019 (LNCS, Vol. 11427)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 372–378. https://doi.org/10.1007/978-3-030-17462-0_24
Thomas R. Shiple, James H. Kukula, and Rajeev K. Ranjan. 1998. A Comparison of Presburger Engines for EFSM Reachability. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1427)*. Springer, 280–292. https://doi.org/10.1007/BFb0028752
Michael Sipser. 2006. *Introduction to Theory of Computation*. Vol. 2. Thomson Course Technology Boston. https://doi.org/10.1145/230514.571645
Randy Smith, Cristian Estan, and Somesh Jha. 2008a. XFA: Faster Signature Matching with Extended Automata. In *IEEE Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/SP.2008.14
Randy Smith, Cristian Estan, Somesh Jha, and Ida Siahaan. 2008b. Fast Signature Matching Using Extended Finite Automaton (XFA). In *ICISS'08 (LNCS, Vol. 5352)*. Springer, 158–172. https://doi.org/10.1007/978-3-540-89862-7_15
Henry Spencer. 1994. Software Solutions in C. Academic Press Professional, Inc., San Diego, CA, USA, Chapter A Regular-expression Matcher, 35–71. http://dl.acm.org/citation.cfm?id=156626.184689
Michael Sperberg-McQueen. [n.d.]. *Notes on finite state automata with counters.* https://www.w3.org/XML/2004/05/msm-cfa.html Accessed: 2018-08-08.
The Sagan team. [n.d.]. The Sagan Log Analysis Engine. https://quadrantsec.com/sagan_log_analysis_engine/.
Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387
Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. [n.d.]. Automata library. https://pajda.fit.vutbr.cz/ituronova/countingautomata.
Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. *Regex Matching with Counting-Set Automata.* Technical Report MSR-TR-2020-31. Microsoft. https://doi.org/10.5281/zenodo.3975566
Milan Češka, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. 2018. Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection. In *Proc. of TACAS'18 (LNCS, Vol. 10806)*. Springer. https://doi.org/10.1007/978-3-319-89963-3_9

Margus Veanes, Peli de Halleux, and Nikolai Tillmann. 2010. Rex: Symbolic Regular Expression Explorer. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. 498–507. https://doi.org/10.1109/ICST.2010.15

Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. 2010. Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams. In *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, Berlin, Heidelberg, 58–78. https://doi.org/10.1007/978-3-642-15512-3_4

Form Methods Syst Des (2012) 41:83–106
DOI 10.1007/s10703-012-0150-8

Forest automata for verification of heap manipulation

Peter Habermehl · Lukáš Holík · Adam Rogalewicz · Jiří Šimáček · Tomáš Vojnar

Published online: 11 April 2012

Abstract We consider verification of programs manipulating dynamic linked data structures such as various forms of singly and doubly-linked lists or trees. We consider important properties for this kind of systems like no null-pointer dereferences, absence of garbage, shape properties, etc. We develop a verification method based on a novel use of tree automata to represent heap configurations. A heap is split into several "separated" parts such that each of them can be represented by a tree automaton. The automata can refer to each other allowing the different parts of the heaps to mutually refer to their boundaries. Moreover, we allow for a hierarchical representation of heaps by allowing alphabets of the tree automata to contain other, nested tree automata. Program instructions can be easily encoded as operations on our representation structure. This allows verification of programs based on symbolic state-space exploration together with refinable abstraction within the so-called abstract regular tree model checking. A motivation for the approach is to combine advantages of automata-based approaches (higher generality and flexibility of the abstraction) with some advantages of separation-logic-based approaches (efficiency). We have implemented our approach and tested it successfully on multiple non-trivial case studies.

This work was supported by the Czech Science Foundation (projects P103/10/0306, P201/09/P531, and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the EU/Czech IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070, the internal BUT project FIT-S-12-1, and the French ANR-09-SEGI project Veridyc.

P. Habermehl
LIAFA, CNRS, Université Paris Diderot, Sorbonne Paris Cité, France

L. Holík · A. Rogalewicz (✉) · J. Šimáček · T. Vojnar
FIT, Brno University of Technology, Brno, Czech Republic
e-mail: rogalew@fit.vutbr.cz

T. Vojnar
e-mail: vojnar@fit.vutbr.cz

L. Holík
Uppsala University, Uppsala, Sweden

J. Šimáček
UJF/CNRS/INPG, VERIMAG, Gières, France

Keywords Pointers · Shape analysis · Regular model checking · Tree automata

1 Introduction

We address verification of sequential programs with complex *dynamic linked data structures* such as various forms of singly- and doubly-linked lists (SLL/DLL), possibly cyclic, shared, hierarchical, and/or having different additional (head, tail, data, and the like) pointers, as well as various forms of trees. We in particular consider C pointer manipulation, but our approach can easily be applied to any other similar language. We concentrate on *safety properties* of the considered programs which includes generic properties like absence of null dereferences, double free operations, dealing with dangling pointers, or memory leakage. Furthermore, to check various shape properties of the involved data structures one can use testers, i.e., parts of code which, in case some desired property is broken, lead the control flow to a designated error location.

For the above purpose, we propose a novel approach of representing sets of heaps via *tree automata* (TA). In our representation, a heap is split in a canonical way into several *tree components* whose roots are the so-called *cut-points*. Cut-points are nodes pointed to by program variables or having several incoming edges. The tree components can refer to the roots of each other, and hence they are "separated" much like heaps described by formulae joined by the separating conjunction in separation logic [16]. Using this decomposition, sets of heaps with a bounded number of cut-points are then represented by a new class of automata called *forest automata* (FA) that are basically tuples of TA accepting tuples of trees whose leaves can refer back to the roots of the trees. Moreover, we allow alphabets of FA to contain *nested FA*, leading to a *hierarchical encoding of heaps*, allowing us to represent even sets of heaps with an unbounded number of cut-points (e.g., sets of DLL). Intuitively, a nested FA can describe a part of a heap with a bounded number of cut-points (e.g., a DLL segment), and by using such an automaton as an alphabet symbol an unbounded number of times, heaps with an unbounded number of cut-points are described. Finally, since FA are not closed under union, we work with sets of forest automata, which are an analogy of disjunctive separation logic formulae.

As a nice theoretical feature of our representation, we show that *inclusion* of sets of heaps represented by finite sets of non-nested FA (i.e., having a bounded number of cut-points) is decidable. This covers sets of complex structures like SLL with head/tail pointers. Moreover, we show how inclusion can be safely approximated for the case of nested FA. Further, C program statements manipulating pointers can be easily encoded as operations modifying FA. Consequently, the symbolic verification framework of *abstract regular tree model checking* [6, 7], which comes with automatically refinable abstractions, can be applied.

The proposed approach brings the principle of *local heap manipulation* (i.e., dealing with separated parts of heaps) from separation logic into the world of automata. The motivation is to combine some advantages of using automata and separation logic. Automata provide higher generality and flexibility of the abstraction (see also below) and allow us to leverage the recent advances of efficient use of non-deterministic automata [2, 3]. As further discussed below, the use of separation allows for a further increase in efficiency compared to a monolithic automata-based encoding proposed in [7].

We have implemented our approach in a prototype tool called *Forester* as a `gcc` plug-in. In our current implementation, if nested FA are used, they are provided manually (similar to the use of pre-defined inductive predicates common in works on separation logic). However, we show that Forester can already successfully handle multiple interesting case studies, proving the proposed approach to be very promising.

Related work The area of verifying programs with dynamic linked data structures has been a subject of intense research for quite some time. Many different approaches based on logics, e.g., [4, 8, 11, 13–17, 19, 20], automata [5, 7, 9], upward closed sets [1], and other formalisms have been proposed. These approaches differ in their generality, efficiency, and degree of automation. Due to space restrictions, we cannot discuss all of them here. Therefore, we concentrate on a comparison with the two closest lines of work, namely, the use of automata as described in [7] and the use of separation logic in the works [4, 19] linked with the Space Invader tool. In fact, as is clear from the above, the approach we propose combines some features from these two lines of research.

Compared to [4, 19], our approach is more general in that it allows one to deal with tree-like structures, too. We note that there are other works on separation logic, e.g., [15], that consider tree manipulation, but these are usually semi-automated only. An exception is [11] which automatically handles even tree structures, but its mechanism of synthesising inductive predicates seems quite dependent on the fact that the dynamic linked data structures are built in a "nice" way conforming to the structure of the predicate to be learned (meaning, e.g., that lists are built by adding elements at the end only).[1]

Further, compared to [4, 19], our approach comes with a more flexible abstraction. We are not building on just using some inductive predicates, but we combine a use of our nested FA with an automatically refinable abstraction on the TA that appear in our representation. Thus our analysis can more easily adjust to various cases arising in the programs being verified. An example is dealing with lists of lists where the sublists are of length 0 or 1, which is a quite practical situation [18]. In such cases, the abstraction used in [4, 19] can fail, leading to an infinite computation (e.g., when, by chance, a list of regularly interleaved lists of length 0 or 1 appears) or generate false alarms (when modified to abstract even pointer links of length 1 to a list segment). For us, such a situation is easy to handle without any need to fine-tune the abstraction manually.

Finally, compared with [7], our newly proposed approach is a bit less general. We cannot handle structures such as, e.g., trees with linked leaves. To handle these structures, we would have to introduce into our approach FA nested not just strictly hierarchically but in an arbitrary, possibly cyclic way, which is an interesting subject for future research. On the other hand, our new approach is more scalable than that of [7]. This is due to the fact that the heap representation in [7] is monolithic, i.e., the whole heap is represented by a single tree skeleton over which additional pointer links are expressed using the so-called routing expressions. The new encoding is much more structured, and so the different operations on the heap, corresponding to a symbolic execution of the verified program, typically influence only small parts of the encoding and not all (or most) of it. The monolithic encoding of [7] has also problems with deletion of elements inside data structures since the routing expressions are built over a tree backbone that is assumed not to change (and hence deleted elements inside data structures are always kept, just marked as deleted). Moreover, the encoding of [7] has troubles with detection of memory leakage, which is in theory possible, but it is so complex that it has never been implemented.

Plan of the paper In Sect. 2, we provide an informal introduction to our proposal of hierarchical forest automata and their use for encoding sets of heaps. In Sect. 3, the notion of (non-hierarchical) forest automata is formalised, and we examine properties of forest automata from the point of view of inclusion checking. Subsequently, Sect. 4 generalises the

[1] We did not find an available implementation of [11], and so we could not try it out ourselves.

notion of forest automata to hierarchical forest automata. In Sect. 5, we propose a verification procedure based on hierarchical forest automata. Section 6 provides a brief description of the Forester tool implementing the proposed approach as well as results obtained from experiments with Forester. Finally, Sect. 7 concludes the paper.

2 From heaps to forests

In this section, we outline in an informal way our proposal of hierarchical forest automata and the way how sets of heaps can be represented by them. For the purpose of the explanation, *heaps* may be viewed as oriented graphs whose nodes correspond to allocated memory cells and edges to pointer links between these cells. The nodes may be labelled by non-pointer data stored in them (assumed to be from a finite data domain) and by program variables pointing to the nodes. Edges may be labelled by the corresponding selectors.

In what follows, we restrict ourselves to *garbage free heaps* in which all memory cells are reachable from pointer variables by following pointer links. However, this is not a restriction in practice since the emergence of garbage can be checked for each executed program statement. If some garbage arises, an error message can be issued and the symbolic computation stopped. Alternatively, the garbage can be removed and the computation continued.

It is easy to see that each heap graph can be *decomposed* into a set of *tree components* when the leaves of the tree components are allowed to reference back to the roots of these components. Moreover, given a total ordering on program variables and selectors, each heap graph may be decomposed into a tuple of tree components in a *canonical way* as illustrated in Fig. 1(a) and (b). In particular, one can first identify the so-called *cut-points*, i.e., nodes that are either pointed to by a program variable or that have several incoming edges. Next, the cut-points can be canonically numbered using a depth-first traversal of the heap graph starting from nodes pointed to by program variables in the order derived from the order of the program variables and respecting the order of selectors. Subsequently, one can split the heap graph into tree components rooted at particular cut-points. These components should

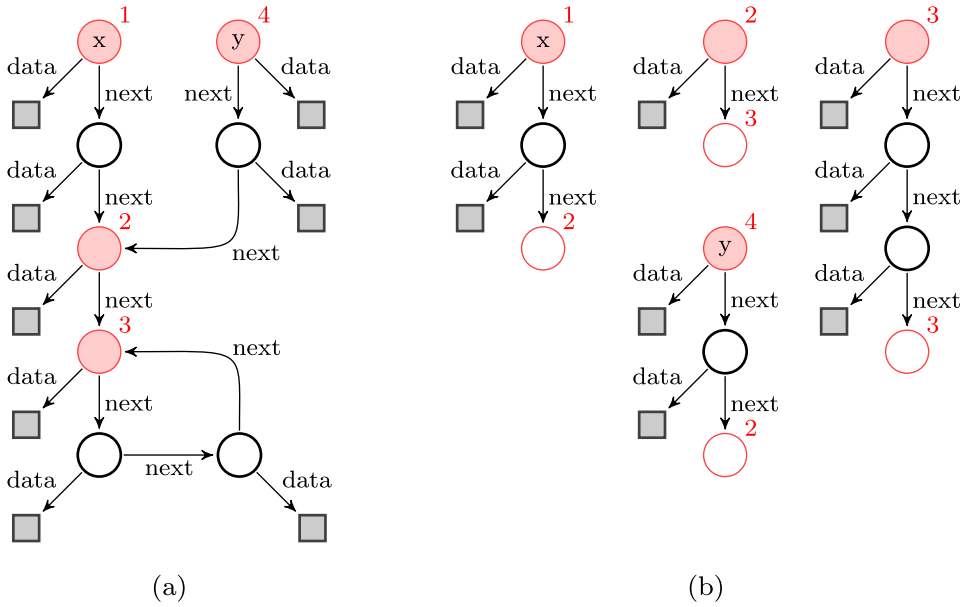


Fig. 1 (**a**) A heap graph with cut-points highlighted in red, (**b**) the canonical tree decomposition of the heap with x ordered before y

contain all the nodes reachable from their root while not passing through any cut-point, plus a copy of each reachable cut-point, labelled by its number. Finally, the tree components can then be canonically ordered according to the numbers of the cut-points representing their roots.

Our proposal of forest automata builds upon the described decomposition of heaps into tree components. In particular, a *forest automaton* (FA) is basically a tuple of tree automata (TA). Each of the tree automata accepts trees whose leaves may refer back to the roots of any of these trees. An FA then represents exactly the set of heaps that may be obtained by taking a single tree from the language of each of the component TA and by gluing the roots of the trees with the leaves referring to them.

Below, we will mostly concentrate on a subclass of FA that we call *canonicity respecting forest automata* (CFA). CFA encode sets of heaps decomposed in a canonical way, i.e., such that if we take any tuple of trees accepted by the given CFA, construct a heap from them, and then canonically decompose it, we get the tuple of trees we started with. This means that in the chosen tuple there is no tree with a root that does not correspond to a cut-point and that the trees are ordered according to the depth-first traversal as described above. The canonicity respecting form allows us to test inclusion on the sets of heaps represented by CFA by testing inclusion component-wise on the languages of the TA constituting the given CFA.

Note, however, that FA are not closed under union. Even for FA having the same number of components, uniting the TA component-wise may yield an FA overapproximating the union of the sets of heaps represented by the original FA (cf. Sect. 3). Thus, we represent unions of FA explicitly as *sets of FA* (SFA), which is similar to dealing with disjunctions of conjunctive separation logic formulae. However, as we will see, inclusion on the sets of heaps represented by SFA is still easily decidable.

The described encoding allows one to represent sets of heaps with a bounded number of cut-points. However, to handle many common dynamic data structures, one needs to represent sets of heaps with an *unbounded number of cut-points*. Indeed, for instance, in doubly-linked lists (DLLs), every node is a cut-point. We solve this problem by representing heaps in a *hierarchical way*. In particular, we collect sets of repeated subgraphs (called *components*) containing cut-points in the so-called *boxes*. Every occurrence of such components can then be replaced by a single edge labelled by the appropriate box. To specify how a subgraph enclosed within a box is connected to the rest of the graph, the subgraph is equipped with the so-called input and output ports. The source vertex of a box then matches the input port of the subgraph, and the target vertex of the edge matches the output port.[2] In this way, a set of heap graphs with an unbounded number of cut-points can be transformed into a set of *hierarchical heap graphs* with a bounded number of cut-points at each level of the hierarchy. Figures 2(a) and (b) illustrate how this approach can basically reduce DLLs into singly-linked lists (with a DLL segment used as a kind of meta-selector).

In general, we allow a box to have more than one output port. Boxes with multiple output ports, however, reduce heap graphs not to graphs but *hypergraphs* with *hyperedges* having a single source node, but multiple target nodes. This situation is illustrated on a simple example shown in Fig. 3. The tree with linked brothers from Fig. 3(a) is turned into a hypergraph with binary hyperedges shown in Fig. 3(c) using the box B from Fig. 3(b). The subgraph encoded by the box B can be connected to its surroundings via its input port i and *two* output

[2]Later on, the term input port will be used to refer to the nodes pointed to by program variables too since these nodes play a similar role as the inputs of components.

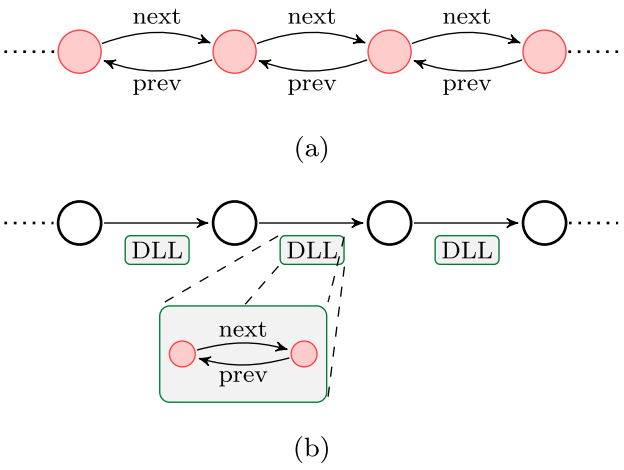


Fig. 2 (**a**) A part of a DLL, (**b**) a hierarchical encoding of the DLL

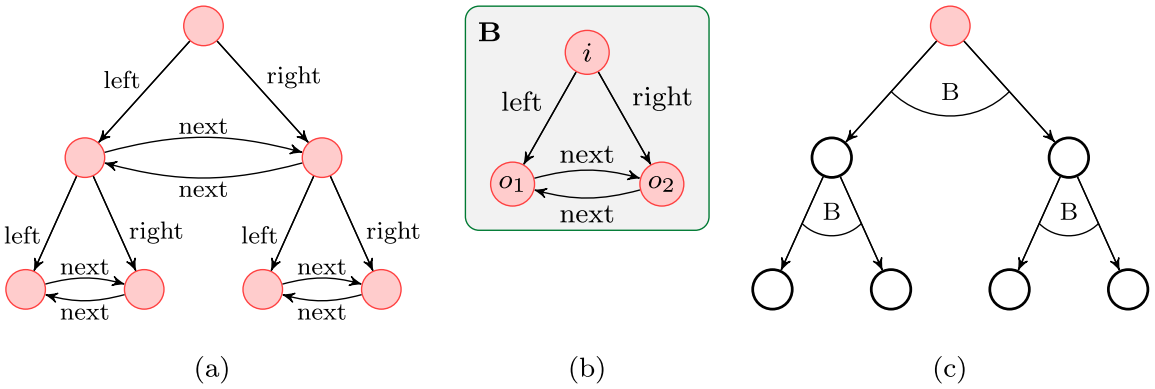


Fig. 3 (**a**) A tree with linked brother nodes, (**b**) a pattern that repeats in the structure and that is linked in such a way that all nodes in the structure are cut-points, (**c**) the tree with linked brother nodes represented using hyperedges labelled by the box B

ports $o1$, $o2$. Therefore, the hypergraph from Fig. 3(c) encodes it by a hyperedge with one source and *two* target nodes.

Sets of heap hypergraphs corresponding either to the top level of the representation or to boxes of different levels can then be decomposed into (hyper)tree components and represented using *hierarchical FA* whose alphabet can contain nested FA.[3] Intuitively, FA appearing in the alphabet of some superior FA play a role similar to that of inductive predicates in separation logic.[4] We restrict ourselves to automata that form a finite and strict hierarchy (i.e., there is no circular use of the automata in their alphabets).

The question of deciding inclusion on sets of heaps represented by hierarchical FA remains open. However, we propose a *canonical decomposition of hierarchical hypergraphs* allowing inclusion to be decided for sets of heap hypergraphs represented by FA provided that the nested FA labelling hyperedges are taken as atomic alphabet symbols. Note that this decomposition is by far not the same as for non-hierarchical heap graphs due to a need

[3]Since graphs are a special case of hypergraphs, in the following, we will work with hypergraphs only. Moreover, to simplify the definitions, we will work with hyperedge-labelled hypergraphs only. Node labels mentioned above will be put at specially introduced nullary hyperedges leaving from the nodes whose label is to be represented.

[4]For instance, we use a nested FA to encode a DLL segment of length 1. In separation logic, the corresponding induction predicate would represent segments of length 1 or more. In our approach, the repetition of the segment is encoded in the structure of the top-level FA.

to deal with nodes that are not reachable on the top level, but are reachable through edges hidden in some boxes. This result allows us to safely approximate inclusion checking on hierarchically represented heaps, which appears to work quite well in practice.

3 Hypergraphs and their representation

We now formalise the notion of hypergraphs and forest automata.

3.1 Hypergraphs

A *ranked alphabet* is a finite set Γ of symbols associated with a map $\# : \Gamma \to \mathbb{N}$. The value $\#(a)$ is called the *rank* of $a \in \Gamma$. We use $\#(\Gamma)$ to denote the maximum rank of a symbol in Γ. A ranked alphabet Γ is a *hypergraph alphabet* if it is associated with a total ordering $\preceq_\Gamma$ on its symbols. For the rest of the section, we fix a hypergraph alphabet Γ.

An (oriented, Γ-labelled) *hypergraph* (with designated input/output ports) is a tuple $G = (V, E, P)$ where:

- V is a finite set of *vertices*.
- E is a finite set of *hyperedges* such that every hyperedge $e \in E$ is of the form $(v, a, (v_1, \ldots, v_n))$ where $v \in V$ is the *source* of e, $a \in \Gamma$, $n = \#(a)$, and $v_1, \ldots, v_n \in V$ are *targets* of e and a-*successors* of v.
- P is the so-called *port specification* that consists of a set of *input ports* $I_P \subseteq V$, a set of *output ports* $O_P \subseteq V$, and a total ordering $\preceq_P$ on $I_P \cup O_P$.

We use $\bar{v}$ to denote a sequence $v_1, \ldots, v_n$ and $\bar{v}.i$ to denote its ith vertex v_i. For symbols $a \in \Gamma$ with $\#(a) = 0$, we write $(v, a) \in E$ to denote that $(v, a, ()) \in E$. Such hyperedges may simulate labels assigned to vertices.

A *path* in a hypergraph $G = (V, E, P)$ is a sequence $\langle v_0, a_1, v_1, \ldots, a_n, v_n \rangle$, $n \geq 0$, where for all $1 \leq i \leq n$, v_i is an a_i-successor of v_{i-1}. G is called *deterministic* iff $\forall (v, a, \bar{v}), (v, a', \bar{v}') \in E$: $a = a' \implies \bar{v} = \bar{v}'$. G is called *well-connected* iff each node $v \in V$ is reachable through some path from some input port of G.

As we have already mentioned in Sect. 2, in hypergraphs representing heaps, input ports correspond to nodes pointed to by program variables or to input nodes of components, and output ports correspond to output nodes of components. Figure 1(a) shows a hypergraph with two input ports corresponding to the variables `x` and `y`. The hyperedges are labelled by selectors `data` and `next`. All the hyperedges are of arity 1. A simple example of a hypergraph with hyperedges of arity 2 is given in Fig. 3(c).

3.2 A forest representation of hypergraphs

We will now define the forest representation of hypergraphs. For that, we will first define a notion of a tree as a basic building block of forests. We will define trees much like hypergraphs but with a restricted shape and without input/output ports. The reason for the latter is that the ports of forests will be defined on the level of the forests themselves, not on the level of the trees that they are composed of.

Formally, an (unordered, oriented, Γ-labelled) *tree* $T = (V, E)$ consists of a set of vertices and hyperedges defined as in the case of hypergraphs with the following additional requirements: (1) V contains a single node with no incoming hyperedge (called the *root* of T and denoted $root(T)$). (2) All other nodes of T are reachable from $root(T)$ via some path.

(3) Each node has at most one incoming hyperedge. (4) Each node appears at most once among the target nodes of its incoming hyperedge (if it has one). Given a tree, we call its nodes with no successors *leaves*.

Let us assume that $\Gamma \cap \mathbb{N} = \emptyset$. An (ordered, Γ-labelled) *forest* (with designated input/output ports) is a tuple $F = (T_1, \ldots, T_n, R)$ such that:

- For every $i \in \{1, \ldots, n\}$, $T_i = (V_i, E_i)$ is a tree that is labelled by the alphabet $(\Gamma \cup \{1, \ldots, n\})$.
- R is a (forest) port specification consisting of a set of *input ports* $I_R \subseteq \{1, \ldots, n\}$, a set of *output ports* $O_R \subseteq \{1, \ldots, n\}$, and a total ordering $\preceq_R$ of $I_R \cup O_R$.
- For all $i, j \in \{1, \ldots, n\}$, (1) if $i \neq j$, then $V_i \cap V_j = \emptyset$, (2) $\#(i) = 0$, and (3) a vertex v with $(v, i) \in E_j$ is not a source of any other edge (it is a leaf). We call such vertices *root references* and denote by $rr(T_i)$ the set of all root references in T_i, i.e., $rr(T_i) = \{v \in V_i \mid (v, k) \in E_i, k \in \{1, \ldots, n\}\}$.

A forest $F = (T_1, \ldots, T_n, R)$ *represents* the hypergraph $\bigotimes F$ obtained by uniting the trees $T_1, \ldots, T_n$ and interconnecting their roots with the corresponding root references. In particular, for every root reference $v \in V_i$, $i \in \{1, \ldots, n\}$, hyperedges leading to v are redirected to the root of T_j where $(v, j) \in E_i$, and v is removed. The sets I_R and O_R then contain indices of the trees whose roots are to be input/output ports of $\bigotimes F$, respectively. Finally, their ordering $\preceq_P$ is defined by the $\preceq_R$-ordering of the indices of the trees whose roots they are. Formally, $\bigotimes F = (V, E, P)$ where:

- $V = \bigcup_{i=1}^{n} V_i \setminus rr(T_i)$, $E = \bigcup_{i=1}^{n} \{(v, a, \bar{v}') \mid a \in \Gamma \wedge \exists (v, a, \bar{v}) \in E_i \ \forall 1 \leq j \leq \#(a) :$ if $\exists (\bar{v}.j, k) \in E_i$ with $k \in \{1, \ldots, n\}$, then $\bar{v}'.j = root(T_k)$, else $\bar{v}'.j = \bar{v}.j\}$,
- $I_P = \{root(T_i) \mid i \in I_R\}$, $O_P = \{root(T_i) \mid i \in O_R\}$,
- $\forall u, v \in I_P \cup O_P$ such that $u = root(T_i)$ and $v = root(T_j)$: $u \preceq_P v \iff i \preceq_R j$.

3.3 Minimal and canonical forests

We now define the canonical form of a forest which will be important later for deciding language inclusion on forest automata, acceptors of sets of hypergraphs.

We call a forest $F = (T_1, \ldots, T_n, R)$ representing the well-connected hypergraph $\bigotimes F$ *minimal* iff the roots of the trees $T_1, \ldots, T_n$ correspond to the *cut-points* of $\bigotimes F$, i.e., those nodes that are either ports, have more than one incoming hyperedge in $\bigotimes F$, or appear more than once as a target of some hyperedge. A minimal forest representation of a hypergraph is unique up to permutations of $T_1, \ldots, T_n$.

In order to get a truly unique canonical forest representation of a well-connected *deterministic* hypergraph $G = (V, E, P)$, it remains to canonically order the trees in its minimal forest representation. To do this, we use the total ordering $\preceq_P$ on ports P and the total ordering $\preceq_\Gamma$ on hyperedge labels Γ of G. We then order the trees according to the order in which their roots are visited in a depth-first traversal (DFT) of G. If all nodes are not reachable from a single port, a series of DFTs is used. The DFTs are started from the input ports in I_P in the order given by $\preceq_P$. During the DFTs, a priority is given to the hyperedges that are smaller in $\preceq_\Gamma$. A canonical representation is obtained this way since we consider G to be deterministic.

Figure 1(b) shows a forest decomposition of the heap graph of Fig. 1(a). The nodes pointed to by variables are input ports of the heap graph. Assuming that the ports are ordered such that the port pointed by x precedes the one pointed by y, then the forest of Fig. 1(b) is a canonical representation of the heap graph of Fig. 1(a).

3.4 Tree automata

Next, we will work towards defining forest automata as tuples of tree automata encoding sets of forests and hence sets of hypergraphs. We start by classical definitions of tree automata and their languages.

Ordered trees Let ϵ denote the empty sequence. An *ordered tree* t over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ satisfying the following conditions: (1) $dom(t)$ is a finite, prefix-closed subset of $\mathbb{N}^*$, and (2) for each $p \in dom(t)$, if $\#(t(p)) = n \geq 0$, then $\{i \mid pi \in dom(t)\} = \{1, \ldots, n\}$. Each sequence $p \in dom(t)$ is called a *node* of t. For a node p, the ith *child* of p is the node pi, and the ith *subtree* of p is the tree t' such that $t'(p') = t(pip')$ for all $p' \in \mathbb{N}^*$. A *leaf* of t is a node p with no children, i.e., there is no $i \in \mathbb{N}$ with $pi \in dom(t)$. Let $\mathbb{T}(\Sigma)$ be the set of all ordered trees over Σ.

Tree automata A (finite, non-deterministic, bottom-up) *tree automaton* (abbreviated as TA in the following) is a quadruple $\mathcal{A} = (Q, \Sigma, \Delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is a ranked alphabet, and Δ is a set of transition rules. Each transition rule is a triple of the form $((q_1, \ldots, q_n), f, q)$ where $n \geq 0$, $q_1, \ldots, q_n, q \in Q$, $f \in \Sigma$, and $\#(f) = n$. We use $f(q_1, \ldots, q_n) \to q$ to denote that $((q_1, \ldots, q_n), f, q) \in \Delta$. In the special case where $n = 0$, we speak about the so-called *leaf rules*.

A *run* of $\mathcal{A}$ over a tree $t \in \mathbb{T}(\Sigma)$ is a mapping $\pi : dom(t) \to Q$ such that, for each node $p \in dom(t)$ where $q = \pi(p)$, if $q_i = \pi(pi)$ for $1 \leq i \leq n$, $t(p)(q_1, \ldots, q_n) \to q$. We write $t \xrightarrow{\pi} q$ to denote that π is a run of $\mathcal{A}$ over t such that $\pi(\epsilon) = q$. We use $t \Longrightarrow q$ to denote that $t \xrightarrow{\pi} q$ for some run π. The *language* of a state q is defined by $L(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of $\mathcal{A}$ is defined by $L(\mathcal{A}) = \bigcup_{q \in F} L(q)$.

3.5 Forest automata

We will now define forest automata as tuples of tree automata extended by a port specification. Tree automata accept trees that are ordered and node-labelled. Therefore, in order to be able to use forest automata to encode sets of forests, we must define a conversion between ordered, node-labelled trees and our unordered, edge-labelled trees.

We convert a deterministic Γ-labelled unordered tree T into a node-labelled ordered tree $ot(T)$ by (1) transferring the information about labels of edges of a node into the symbol associated with the node and by (2) ordering the successors of the node. More concretely, we label each node of the ordered tree $ot(T)$ by the set of labels of the hyperedges leading from the corresponding node in the original tree T. Successors of the node in $ot(T)$ correspond to the successors of the original node in T, and are ordered w.r.t. the order $\preceq_\Gamma$ of hyperedge labels through which the corresponding successors are reachable in T (while always keeping tuples of nodes reachable via the same hyperedge together, ordered in the same way as they were ordered within the hyperedge). The rank of the new node label is given by the sum of ranks of the original hyperedge labels embedded into it. Below, we use Σ_Γ to denote the ranked node alphabet obtained from Γ as described above.

The notion of forest automata A *forest automaton* over Γ (with designated input/output ports) is a tuple $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ where:

- For all $1 \leq i \leq n$, $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$ is a TA with $\Sigma = \Sigma_\Gamma \cup \{1, \ldots, n\}$ and $\#(i) = 0$.
- R is defined as for forests, i.e., it consists of input and output ports $I_R, O_R \subseteq \{1, \ldots, n\}$ and a total ordering $\preceq_R$ on $I_R \cup O_R$.

The *forest language* of $\mathcal{F}$ is the set of forests $L_F(\mathcal{F}) = \{(T_1, \ldots, T_n, R) \mid \forall 1 \leq i \leq n : ot(T_i) \in L(\mathcal{A}_i)\}$, i.e., the forest language is obtained by taking the Cartesian product of the tree languages, unordering the trees that appear in its elements, and extending them by the port specification. The forest language of $\mathcal{F}$ in turn defines the *hypergraph language* of $\mathcal{F}$ which is the set of hypergraphs $L(\mathcal{F}) = \{\bigotimes F \mid F \in L_F(\mathcal{F})\}$.

An FA $\mathcal{F}$ *respects canonicity* iff for each forest $F \in L_F(\mathcal{F})$, the hypergraph $\bigotimes F$ is well-connected, and F is its canonical representation. We abbreviate canonicity respecting FA as CFA. It is easy to see that comparing sets of hypergraphs represented by CFA can be done *component-wise* as described in the below proposition.

Proposition 1 *Let* $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ *and* $\mathcal{F}' = (\mathcal{A}'_1, \ldots, \mathcal{A}'_m, R')$ *be two CFA. Then,* $L(\mathcal{F}) \subseteq L(\mathcal{F}')$ *iff* $n = m$, $R = R'$, *and* $\forall 1 \leq i \leq n : L(\mathcal{A}_i) \subseteq L(\mathcal{A}'_i)$.

3.6 Transforming FA into canonicity respecting FA

In order to facilitate inclusion checking, each FA can be algorithmically transformed (split) into a finite set of CFA such that the union of their languages equals the original language. We describe the transformation in a more detailed way below.

First, we label the states of the component TA of the given FA by special labels. For each state, these labels capture all possible orders in which root references appear in the leaves of the trees accepted at this state when the left-most (i.e., the first) appearance of each root-reference is considered only. Moreover, the labels capture which of the references appear multiple times. Intuitively, following the first appearances of the root references in the leaves of tree components is enough to see how a depth first traversal through the represented hypergraph orders the roots of the tree components. The knowledge of multiple references to the same root from a single tree is then useful for checking which nodes should really be the roots.

The computed labels are subsequently used to possibly split the given FA into several FA such that the accepting states of the component TA of each of the obtained FA are labelled in a unique way. This guarantees that the obtained FA are canonicity respecting up to the fact that the roots of some of the trees accepted by component TA need not be cut-points (and up to the ordering of the component TA). Thus, subsequently, some of the TA may get merged. Finally, we order the remaining component TA in a way consistent with the DFT ordering on the cut-points of the represented hypergraphs (which after the splitting is the same for all the hypergraphs represented by each obtained FA). To order the component TA, the labels of the accepting states can be conveniently used.

More precisely, consider a forest automaton $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$, $n \geq 1$, and any of its component tree automata $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$, $1 \leq i \leq n$. We label each state $q \in Q_i$ by a set of labels (w, Y), $w \in \{1, \ldots, n\}^*$, $Y \subseteq \{1, \ldots, n\}$, for which there is a tree $t \in L(q)$ such that

- w is the string that records the order in which root references appear for the first time in the leaves of t (i.e., w is the concatenation of the labels of the leaves labelled by root references, restricted to the first occurrence of each root reference), and
- Y is the set of root references that appear more than once in the leaves of t.

Such labelling can be obtained by first labelling states w.r.t. the leaf rules and then propagating the so-far obtained labels bottom-up. If the final states of $\mathcal{A}_i$ get labelled by several different labels, we make a copy of the automaton for each of these labels, and in each of them, we preserve only the transitions that allow trees with the appropriate label of the root

to be accepted.[5] This way, all the component automata can be processed and then new forest automata can be created by considering all possible combinations of the transformed TA.

Clearly, each of the FA created above represents a set of hypergraphs that have the same number of cut-points (corresponding either to ports, nodes referenced at least twice from a single component tree, or referenced from several component trees) that get ordered in the same way in the depth first traversal of the hypergraphs. However, it may be the case that some roots need not correspond to cut-points. This is easy to detect by looking for a root reference that does not appear in the set part of any label of some final state and that does not appear in the labels of two different component tree automata. A useless root can then be eliminated by adding transition rules of the appropriate component tree automaton $\mathcal{A}_i$ to those of the tree automaton $\mathcal{A}_j$ that refers to that root and by gluing final states of $\mathcal{A}_i$ with the states of $\mathcal{A}_j$ accepting the root reference i.

It remains to order the component TA within each of the obtained FA in a way consistent with the DFT ordering of the cut-points of the represented hypergraphs (which is now the same for all the hypergraphs represented by a single FA due to the performed splitting). To order the component TA of any of the obtained FA, one can use the w-part of the labels of its accepting states. One can then perform a DFT on the component TA, considering the TA as atomic objects. One starts with the TA that accept trees whose roots represent ports and processes them wrt. the ordering of ports. When processing a TA $\mathcal{A}$, one considers as its successors the TA that correspond to the root references that appear in the w-part of the labels of the accepting states of $\mathcal{A}$. Moreover, the successor TA are processed in the order in which they are referenced from the labels. When the DFT is over, the component TA may get reordered according to the order in which they were visited.

Subsequently, the port specification R and root references in leaves must be updated to reflect the reordering. If the original sets I_R or O_R contain a port i, and the ith tree was moved to the jth position, then i must be substituted by j in I_R, O_R, and $\preceq_R$ as well as in all root references. This finally leads to a set of canonicity respecting FA.

Note that, in practice, it is not necessary to tightly follow the above described process. Instead, one can arrange the symbolic execution of statements in such a way that when starting with a CFA, one obtains an FA which already meets some requirements for CFA. Most notably, the splitting of component TA—if needed—can be efficiently done already during the symbolic execution of the particular statements. Therefore, transforming an FA obtained this way into the corresponding CFA involves the elimination of redundant roots and the root reordering only.

3.7 Sets of forest automata

The class of languages of FA (and even CFA) is not closed under union since a forest language of a FA corresponds to the Cartesian product of the languages of all its components, and not every union of Cartesian products may be expressed as a single Cartesian product. For instance, consider two CFA $\mathcal{F} = (\mathcal{A}, \mathcal{B}, R)$ and $\mathcal{F}' = (\mathcal{A}', \mathcal{B}', R)$ such that $L_F(\mathcal{F}) = \{(a, b, R)\}$ and $L_F(\mathcal{F}') = \{(c, d, R)\}$ where a, b, c, d are distinct trees. The forest language of the FA $(\mathcal{A} \cup \mathcal{A}', \mathcal{B} \cup \mathcal{B}', R)$ is $\{(x, y, R) \mid (x, y) \in \{a, c\} \times \{b, d\}\}$, and there is no FA with the hypergraph language equal to $L(\mathcal{F}) \cup L(\mathcal{F}')$.

[5] More technically, given a labelled TA, one can first make a separate copy of each state for each of its labels, connect the states by transitions such that the obtained singleton labelling is respected, then make a copy of the TA for each label of accepting states, and keep the accepting status for a single labelling of accepting states in each of the copies only.

Due to the above, we cannot transform a set of CFA obtained by canonising a given FA into a single CFA. Likewise, when we obtain several CFA when symbolically executing several program paths leading to the same program location, we cannot merge them into a single CFA without risking a loss of information. Consequently, we will explicitly work with *finite sets of (canonicity-respecting) forest automata*, S(C)FA for short, where the language $L(\mathcal{S})$ of a finite set $\mathcal{S}$ of FA is defined as the union of the languages of its elements. This, however, means that we need to be able to decide language inclusion on SFA.

Testing inclusion on SFA The problem of checking inclusion on SFA, this is, checking whether $L(\mathcal{S}) \subseteq L(\mathcal{S}')$ where $\mathcal{S}, \mathcal{S}'$ are SFA, can be reduced to a problem of checking inclusion on tree automata. We may w.l.o.g. assume that $\mathcal{S}$ and $\mathcal{S}'$ are SCFA.

We will transform every FA $\mathcal{F}$ in $\mathcal{S}$ and $\mathcal{S}'$ into a TA $\mathcal{A}^{\mathcal{F}}$ which accepts the language of trees where:

- The root of each of these trees is labelled by a special fresh symbol (parameterised by n and the port specification of $\mathcal{F}$).
- The root has n children, one for each tree automaton of $\mathcal{F}$.
- For each $1 \leq i \leq n$, the ith child of the root is the root of a tree accepted by the ith tree automaton of $\mathcal{F}$.

Trees accepted by $\mathcal{A}^{\mathcal{F}}$ are therefore unique encodings of hypergraphs in $L(\mathcal{F})$. We will then test the inclusion $L(\mathcal{S}) \subseteq L(\mathcal{S}')$ by testing the tree automata language inclusion between the union of TA obtained from $\mathcal{S}$ and the union of TA obtained from $\mathcal{S}'$.

Formally, let $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ be an FA where $\mathcal{A}_i = (\Sigma, Q_i, \Delta_i, F_i)$ for each $1 \leq i \leq n$. Without a loss of generality, assume that $Q_i \cap Q_j = \emptyset$ for each $1 \leq i < j \leq n$. We define the TA $\mathcal{A}^{\mathcal{F}} = (\Sigma \cup \{\curlywedge_n^R\}, Q, \Delta, \{q^{top}\})$ where:

- $\curlywedge_n^R \notin \Sigma$ is a fresh symbol with $\#(\curlywedge_n^R) = n$,
- $q^{top} \notin \bigcup_{i=1}^{n} Q_i$ is a fresh accepting state,
- $Q = \bigcup_{i=1}^{n} Q_i \cup \{q^{top}\}$, and
- $\Delta = \bigcup_{i=1}^{n} \Delta_i \cup \Delta^{top}$ where Δ^{top} contains the rule $\curlywedge_n^R(q_1, \ldots, q_n) \to q^{top}$ for each $(q_1, \ldots, q_n) \in F_1 \times \cdots \times F_n$.

It is now easy to see that the following proposition holds (in the proposition, "$\cup$" stands for the usual tree automata union).

Proposition 2 *For SCFA $\mathcal{S}$ and $\mathcal{S}'$, $L(\mathcal{S}) \subseteq L(\mathcal{S}') \iff L(\bigcup_{\mathcal{F} \in \mathcal{S}} \mathcal{A}^{\mathcal{F}}) \subseteq L(\bigcup_{\mathcal{F}' \in \mathcal{S}'} \mathcal{A}^{\mathcal{F}'})$.*

4 Hierarchical hypergraphs

As discussed informally in Sect. 2, simple forest automata cannot express sets of data structures with unbounded numbers of cut-points like, e.g., the set of all doubly-linked lists or the set of all trees with linked brothers (Figs. 2 and 3). To capture such data structures, we will enrich the expressive power of forest automata by allowing them to be hierarchically nested. For the rest of the section, we fix a hypergraph alphabet Γ.

4.1 Hierarchical hypergraphs, components, and boxes

We first introduce hypergraphs with hyperedges labelled by the so-called boxes which are sets of hypergraphs (defined up to isomorphism).[6] A hypergraph G with hyperedges labelled by boxes encodes a set of hypergraphs. The hypergraphs encoded by G can be obtained by replacing every hyperedge of G labelled by a box by some hypergraph from the box. The hypergraphs within the boxes may themselves have hyperedges labelled by boxes, which gives rise to a hierarchical structure (which we require to be of a finite depth).

Let Υ be a hypergraph alphabet. First, we define an Υ-labelled *component* as an Υ-labelled hypergraph $C = (V, E, P)$ which satisfies the requirement that $|I_P| = 1$ and $I_P \cap O_P = \emptyset$. Then, an Υ-labelled *box* is a non-empty set B of Υ-labelled components such that all of them have the same number of output ports. This number is called the *rank of the box* B and denoted by $\#(B)$. Let $\mathbb{B}[\Upsilon]$ be the ranked alphabet containing all Υ-labelled boxes such that $\mathbb{B}[\Upsilon] \cap \Upsilon = \emptyset$. The operator $\mathbb{B}$ gives rise to a hierarchy of alphabets $\Gamma_0, \Gamma_1, \ldots$ where:

- $\Gamma_0 = \Gamma$ is the set of *plain symbols*,
- for $i \geq 0$, $\Gamma_{i+1} = \Gamma_i \cup \mathbb{B}[\Gamma_i]$ is the set of *symbols of level* $i + 1$.

A Γ_i-labelled hypergraph H is then called a *Γ-labelled (hierarchical) hypergraph of level i*, and we refer to the Γ_{i-1}-labelled boxes appearing on edges of H as to *nested boxes of* H. A Γ-labelled hypergraph is sometimes called a *plain* Γ-labelled hypergraph.

Semantics of hierarchical hypergraphs A Γ-labelled hierarchical hypergraph H encodes a set $[\![H]\!]$ of plain hypergraphs, called the *semantics* of H. For a set S of hierarchical hypergraphs, we use $[\![S]\!]$ to denote the union of semantics of its elements.

If H is plain, then $[\![H]\!]$ contains just H itself. If H is of level $j > 0$, then hypergraphs from $[\![H]\!]$ are obtained in such a way that hyperedges labelled by boxes $B \in \Gamma_j$ are substituted in all possible ways by plain components from $[\![B]\!]$. The substitution is similar to an ordinary hyperedge replacement used in graph grammars. When an edge e is substituted by a component C, the input port of C is identified with the source node of e, and the output ports of C are identified with the target nodes of e. The correspondence of the output ports of C and the target nodes of e is defined using the order of the target nodes in e and the ordering of ports of C. The edge e is finally removed from H.

Formally, given a Γ-labelled hierarchical hypergraph $H = (V, E, P)$, a hyperedge $e = (v, a, \bar{v}) \in E$, and a component $C = (V', E', P')$ where $\#(a) = |O_{P'}| = k$, the substitution of e by C in H results in the hypergraph $H[C/e]$ defined as follows. Let $o_1 \preceq_P \ldots \preceq_P o_k$ be the ports of O_P ordered by $\preceq_P$. W.l.o.g., assume $V \cap V' = \emptyset$. C will be connected to H by identifying its ports with their matching vertices of e. We define for every vertex $w \in V'$ its matching vertex $match(w)$ such that (1) if $w \in I_{P'}$, $match(w) = v$ (the input port of C matches the source of e), (2) if $w = o_i, 1 \leq i \leq k$, $match(w) = \bar{v}.i$ (the output ports of C match the corresponding targets of e), and (3) $match(w) = w$ otherwise (an inner node of C is not matched with any node of H). Then $H[C/e] = (V'', E'', P)$ where $V'' = V \cup (V' \setminus (I_{P'} \cup O_{P'}))$ and $E'' = (E \setminus \{e\}) \cup \{(v'', a', \bar{v}'') \mid \exists (v', a', \bar{v}') \in E' : match(v') = v'' \wedge \forall 1 \leq i \leq k : match(\bar{v}'.i) = \bar{v}''.i\}$.

[6] Dealing with hypergraphs and later also automata defined up to isomorphism avoids a need to deal with classes instead of sets. We will not repeat this fact later on.

We can now give an inductive definition of $[\![H]\!]$. Let $e_1 = (v_1, B_1, \bar{v}_1), \ldots, e_n = (v_n, B_n, \bar{v}_n)$ be all edges of H labelled by Γ-labelled boxes. Then, $G \in [\![H]\!]$ iff it is obtained from H by successively substituting every e_i by a component $C_i \in [\![B_i]\!]$, i.e.,

$$[\![H]\!] = \{H[C_1/e_1] \ldots [C_n/e_n] \mid C_1 \in [\![B_1]\!], \ldots, C_n \in [\![B_n]\!]\}.$$

Figure 2(b) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)graph of Fig. 2(a). Similarly, Fig. 3(c) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)-graph of Fig. 3(a).

4.2 Hierarchical forest automata

We now define hierarchical forest automata that represent sets of hierarchical hypergraphs. The hierarchical FA are FA whose alphabet can contain symbols which encode boxes appearing on edges of hierarchical hypergraphs. The boxes are themselves represented using hierarchical FA.

To define an alphabet of hierarchical FA, we will take an approach similar to the one used for the definition of hierarchical hypergraphs. First, we define an operator $\mathbb{A}$ which for a hypergraph alphabet Υ returns the ranked alphabet containing the set of all SFA $\mathcal{S}$ over (a finite subset of) Υ such that $L(\mathcal{S})$ is an Υ-labelled box and such that $\mathbb{A}[\Upsilon] \cap \Upsilon = \emptyset$. The rank of $\mathcal{S}$ in the alphabet $\mathbb{A}[\Upsilon]$ is the rank of the box $L(\mathcal{S})$. The operator $\mathbb{A}$ gives rise to a hierarchy of alphabets $\mathbf{\Gamma}_0, \mathbf{\Gamma}_1, \ldots$ where:

- $\mathbf{\Gamma}_0 = \Gamma$ is the set of *plain symbols*,
- for $i \geq 0$, $\mathbf{\Gamma}_{i+1} = \mathbf{\Gamma}_i \cup \mathbb{A}[\mathbf{\Gamma}_i]$ is the set of *symbols of level* $i + 1$.

A hierarchical FA $\mathcal{F}$ over $\mathbf{\Gamma}_i$ is then called a Γ-labelled *(hierarchical) FA of level* i, and we refer to the hierarchical SFA over $\mathbf{\Gamma}_{i-1}$ appearing within alphabet symbols of $\mathcal{F}$ as to *nested SFA of* $\mathcal{F}$.

Let $\mathcal{F}$ be a hierarchical FA. We now define an operator $\sharp$ that translates any $\mathbf{\Gamma}_i$-labelled hypergraph $G = (V, E, P) \in L(\mathcal{F})$ to a Γ-labelled hierarchical hypergraph H of level i (i.e., it translates G by transforming the SFA that appear on its edges to the boxes they represent). Formally, $G^\sharp$ is defined inductively as the Γ-labelled hierarchical hypergraph $H = (V, E', P)$ of level i that is obtained from the hypergraph G by replacing every edge $(v, \mathcal{S}, \bar{v}) \in E$, labelled by a Γ-labelled hierarchical SFA $\mathcal{S}$, by the edge $(v, L(\mathcal{S})^\sharp, \bar{v})$, labelled by the box $L(\mathcal{S})^\sharp$ where $L(\mathcal{S})^\sharp$ denotes the set (box) $\{X^\sharp \mid X \in L(\mathcal{S})\}$. Then, we define the semantics of a hierarchical FA $\mathcal{F}$ over Γ as the set of Γ-labelled (plain) hypergraphs $[\![\ ]\!]F = [\![L(\mathcal{F})^\sharp]\!]$.

Notice that a hierarchical SFA of any level has finitely many nested SFA of a lower level only. Therefore, a hierarchical SFA is a finitely representable object. Notice also that even though the maximum number of cut-points of hypergraphs from $L(\mathcal{S})^\sharp$ is fixed (SFA always accept hypergraphs with a fixed maximum number of cut-points), the number of cut-points of hypergraphs in $[\![\mathcal{S}]\!]$ may be unbounded. The reason is that hypergraphs from $L(\mathcal{S})^\sharp$ may contain an unbounded number of hyperedges labelled by boxes B such that hypergraphs from $[\![B]\!]$ contain cut-points too. These cut-points then appear in hypergraphs from $[\![\mathcal{S}]\!]$, but they are not visible at the level of hypergraphs from $L(\mathcal{S})^\sharp$.

Hierarchical SFA are therefore finite representations of sets of hypergraphs with possibly unbounded numbers of cut-points.

4.3 Inclusion and well-connectedness on hierarchical SFA

In this section, we aim at checking well-connectedness and inclusion of sets of hypergraphs represented by hierarchical FA. Since considering the full class of hierarchical hypergraphs would unnecessarily complicate our task, we enforce a restricted form of hierarchical automata that rules out some rather artificial scenarios and that allows us to handle the automata hierarchically (i.e., using some pre-computed information for nested FA rather than having to unfold the entire hierarchy all the time). In particular, the restricted form guarantees that:

1. For a hierarchical hypergraph H, well-connectedness of hypergraphs in $[\![H]\!]$ is equivalent to the so-called box-connectedness of H. Box-connectedness is a property introduced below that can be easily checked and that basically considers paths from input ports to output ports and vice versa, in the latter case through hyperedges hidden inside nested boxes.
2. Determinism of hypergraphs from $[\![H]\!]$ implies determinism of H.

The two above properties simplify checking inclusion and well-connectedness considerably since for a general hierarchical hypergraph H, well-connectedness of H is neither implied nor it implies well-connectedness of hypergraphs from $[\![H]\!]$. This holds also for determinism. The reason is that a component C in a nested box of H may interconnect its ports in an arbitrary way. It may contain paths from output ports to both input and output ports (including paths from an output port to another output port not passing the input port), but it may be missing paths from the input port to some of the output ports.

Using the above restriction, we will show below a safe approximation of inclusion checking on hierarchical SFA, and we will also show that this approximation is precise in some cases. Despite the introduced restriction, the description is quite technical, and it may be skipped on the first reading. Indeed, it turns out that in practice, an even more aggressive approximation of inclusion checking in which nested boxes are taken as atomic symbols is often sufficient.

Properness and box-connectedness Given a Γ-labelled component C of level 0, we define its *backward reachability set* $br(C)$ as the set of indices i for which there is a path from the i-th output port of C *back* to the input port of C. Given a box B over Γ, we inductively define B to be *proper* iff all its nested boxes are proper, $br(C_1) = br(C_2)$ for any $C_1, C_2 \in [\![B]\!]$, and the following holds for all components $C \in [\![B]\!]$:

1. C is well-connected.
2. If there is a path from the i-th to the j-th output port of C, $i \neq j$, then $i \in br(C)$.[7]

For a proper box B, we use $br(B)$ to denote $br(C)$ for $C \in [\![B]\!]$. A hierarchical hypergraph H is called *well-formed* iff all its nested boxes are proper. In that case, the conditions above imply that either all or no hypergraphs from $[\![H]\!]$ are well-connected and that well-connectedness of hypergraphs in $[\![H]\!]$ may be judged based only on the knowledge of $br(B)$ for each nested box B of H, without a need to reason about the semantics of B (in particular, Point 2 in the above definition of proper boxes guarantees that we do not have to take into account paths that interconnect output ports of B). This is formalised below.

[7]Notice that this definition is correct since boxes of level 0 have no nested boxes, and the recursion stops at them.

Let $H = (V, E, P)$ be a well-formed Γ-labelled hierarchical hypergraph with a set X of nested boxes. We define the *backward reachability graph* of H as the $\Gamma \cup X \cup X^{br}$-labelled hypergraph $H^{br} = (V, E \cup E^{br}, P)$ where $X^{br} = \{(B, i) \mid B \in X \wedge i \in br(B)\}$ and $E^{br} = \{(v_i, (B, i), (v)) \mid B \in X \wedge (v, B, (v_1, \ldots, v_n)) \in E \wedge i \in br(B)\}$. We say that H is *box-connected* iff H^{br} is well-connected. The below proposition clearly holds.

Proposition 3 *If H is a well-formed hierarchical hypergraph, then the hypergraphs from $[\![H]\!]$ are well-connected iff H is box-connected. Moreover, if hypergraphs from $[\![H]\!]$ are deterministic, then both H and H^{br} are deterministic hypergraphs.*

We straightforwardly extend the above notions to hypergraphs with hyperedges labelled by hierarchical SFA, treating these SFA-labels as if they were the boxes they represent. Particularly, we call a hierarchical SFA $\mathcal{S}$ proper iff it represents a proper box $[\![\ ]\!]S$, we let $br(\mathcal{S}) = br([\![\mathcal{S}]\!])$, and for a $\Gamma \cup Y$-labelled hypergraph G where Y is a set of proper SFA, its backward reachability hypergraph G^{br} is defined based on br in the same way as the backward reachability hypergraph of a hierarchical hypergraph above (just instead of boxes, we deal with their SFA representations). We also say that G is box-connected iff G^{br} is well-connected.

Checking properness and well-connectedness We now outline algorithms for checking properness of nested SFA and well-connectedness of SFA.

Properness of nested SFA can be checked relatively easily since we can take advantage of the fact that nested SFA of a proper SFA must be proper as well. We start with nested SFA of level 0 which contain no nested SFA, we check their properness and compute the values of the backward reachability function br for them. To do this we can label TA states similarly to Sect. 3.6. A unique label of each root in the SFA representing the box guarantees that the br function will be equal for all hypergraphs hidden in the box. Then, we iteratively increase the level j and for each j, we check properness of the nested SFA of level j and compute the values of the function br. For this, we use the values of br that we have computed for the nested SFA of level $j - 1$, and we can also take advantage of the fact that the nested SFA of level $j - 1$ have been shown to be proper. We can again use the labels attached to all tree automata states. The difference from level 0 is that we have to extend the labels in order to capture also the backward reachability of the edges labelled by nested SFA.

Now, given an FA $\mathcal{F}$ over Γ with proper nested SFA, we can check well-connectedness of hypergraphs from $[\![\mathcal{F}]\!]$ as follows: (1) for each nested SFA $\mathcal{S}$ of $\mathcal{F}$, we compute like above (and cache for further use) the value $br(\mathcal{S})$, and (2) using this value, we check box-connectedness of hypergraphs in $L(\mathcal{F})$ without a need of reasoning about the inner structure of the nested SFA [12].

The problem of checking inclusion on hierarchical FA Checking inclusion on hierarchical automata over Γ with nested boxes from X, i.e., given two hierarchical FA $\mathcal{F}$ and $\mathcal{F}'$, checking whether $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$, is a hard problem, even under the assumption that nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ are proper. Its decidability is not known. In this paper, we choose a pragmatic approach and give only a semi-algorithm that is efficient and works well in practical cases. The idea is simple. Since the implications $L(\mathcal{F}) \subseteq L(\mathcal{F}') \implies L(\mathcal{F})^\sharp \subseteq L(\mathcal{F}')^\sharp \implies [\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ obviously hold, we may safely approximate the solution of the inclusion problem by deciding whether $L(\mathcal{F}) \subseteq L(\mathcal{F}')$ (i.e., we abstract away the semantics of nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ and treat them as ordinary labels).

From now on, assume that our hierarchical FA represent only deterministic well-connected hypergraphs, i.e., that $[\![\mathcal{F}]\!]$ and $[\![\mathcal{F}']\!]$ contain only well-connected deterministic hypergraphs. Note that this assumption is in particular fulfilled for hierarchical FA representing garbage-free heaps.

We cannot directly use the results on inclusion checking of Sect. 3.5, based on a canonical forest representation and canonicity respecting FA, since they rely on well-connectedness of hypergraphs from $L(\mathcal{F})$ and $L(\mathcal{F}')$, which is now *not* necessarily the case. The reason is that hypergraphs represented by a not well-connected hierarchical hypergraph H can themselves still be well-connected via backward links hidden in boxes. However, by Proposition 3, every hypergraph G from $L(\mathcal{F})$ or $L(\mathcal{F}')$ is box-connected, and both G and G^{br} are deterministic. As we show below, these properties are still sufficient to define a canonical forest representation of G, which in turn yields a canonicity respecting form of hierarchical FA.

Canonicity respecting hierarchical FA Let Y be a set of proper SFA over Γ. We aim at a canonical forest representation $F = (T_1, \ldots, T_n, R)$ of a $\Gamma \cup Y$-labelled hypergraph $G = \bigotimes F$ which is box-connected and such that both G and G^{br} are deterministic. By extending the approach used in Sect. 3.5, this will be achieved via an unambiguous definition of the *root-points* of G, i.e., the nodes of G that correspond to the roots of the trees $T_1, \ldots, T_n$, and their ordering.

The root-points of G are defined as follows. First, every cut-point (port or a node with more than one incoming edge) is a *root-point of Type 1*. Then, every node with no incoming edge is a *root-point of Type 2*. Root-points of Type 2 are entry points of parts of G that are not reachable from root-points of Type 1 (they are only backward reachable). However, not every such part of G has a unique entry point which is a root-point of Type 2. Instead, there might be a simple loop such that there are no edges leading into the loop from outside. To cover a part of G that is reachable from such a loop, we have to choose exactly one node of the loop to be a root-point. To choose one of them unambiguously, we define a total ordering $\preceq_G$ on nodes of G and choose the smallest node wrt. this ordering to be a *root-point of Type 3*. After unambiguously determining all root-points of G, we may order them according to $\preceq_G$, and we are done.

A suitable total ordering $\preceq_G$ on V can be defined taking advantage of the fact that G^{br} is well-connected and deterministic. Therefore, it is obviously possible to define $\preceq_G$ as the order in which the nodes are visited by a deterministic depth-first traversal that starts at input ports. The details on how this may be algorithmically done on the structure of forest automata may be found in [12].

We say that a hierarchical FA $\mathcal{F}$ over Γ with proper nested SFA and such that hypergraphs from $[\![\mathcal{F}]\!]$ are deterministic and well-connected *respects canonicity* iff each forest $F \in L_F(\mathcal{F})$ is a canonical representation of the hypergraph $\bigotimes F$. We abbreviate canonicity respecting hierarchical FA as hierarchical CFA. Analogically as for ordinary CFA, respecting canonicity allows us to compare languages of hierarchical CFA component-wise as described in the below proposition.

Proposition 4 *Let $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ and $\mathcal{F}' = (\mathcal{A}'_1, \ldots, \mathcal{A}'_m, R')$ be hierarchical CFA. Then, $L(\mathcal{F}) \subseteq L(\mathcal{F}')$ iff $n = m$, $R = R'$, and $\forall 1 \leq i \leq n : L(\mathcal{A}_i) \subseteq L(\mathcal{A}'_i)$.*

Proposition 4 allows us to safely approximate inclusion of the sets of hypergraphs encoded by hierarchical FA (i.e., to safely approximate the test $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ for hierarchical FA $\mathcal{F}$, $\mathcal{F}'$). This turns out to be sufficient for all our case studies (cf. Sect. 6). Moreover, the

described inclusion checking is precise at least in some cases as discussed below. A generalisation of the result to sets of hierarchical CFA can be obtained as for ordinary SFA. Hierarchical FA that do not respect canonicity may be algorithmically split into several hierarchical CFA, similarly as ordinary CFA [12].

Precise inclusion on hierarchical FA In many practical cases, approximating the inclusion $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ by deciding $L(\mathcal{F}) \subseteq L(\mathcal{F}')$ is actually precise. A condition that guarantees this is the following:

Condition 1 $\forall H \in L(\mathcal{F})^\sharp \, \forall H' \in L(\mathcal{F}')^\sharp : H \neq H' \implies [\![H]\!] \cap [\![H']\!] = \emptyset$. Intuitively, this means that one cannot have two distinct hierarchical hypergraphs representing the same plain hypergraph.

Clearly, Condition 1 holds if the following two more concrete conditions hold:

Condition 2 Nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ represent a set of boxes X that *do not overlap*.

Condition 3 Every $H \in L(\mathcal{F})^\sharp \cup L(\mathcal{F}')^\sharp$ is *maximally boxed* by boxes from X.

The notions of maximally boxed hypergraphs and non-overlapping boxes are defined as follows. A hierarchical hypergraph H is *maximally boxed* by boxes from a set X iff all its nested boxes are from X, and no part of H can be "hidden" in a box from X, this is, there is no hypergraph G and no component $C \in B$, $B \in X$ such that $G[C/e] = H$ for some edge e of G. Boxes from a set of boxes X over Γ *do not overlap* iff for every hypergraph G over Γ, there is only one hierarchical hypergraph H over Γ which is maximally boxed by boxes from X and such that $G \in [\![H]\!]$.

We note that the boxes represented by the nested SFA that appear in the case studies presented in this paper satisfy Conditions 2 and 3, and so Condition 1 is satisfied too. Hence, inclusion tests performed within our case studies are precise.

5 The verification procedure based on forest automata

We now briefly describe our verification procedure. As already said, we consider sequential, non-recursive C programs manipulating dynamic linked data structures via program statements `x = y`, `x = y->s`, `x = null`, `x->s = y`, `malloc(x)`, and `free(x)` together with pointer and data equality tests and common control flow statements as discussed in more details below.[8] Each allocated cell may have several next pointer selectors and contain data from some finite domain.[9] We use *Sel* to denote the set of all selectors and *Data* to denote the data domain. The cells may be pointed by program variables whose set is denoted as *Var* below.

[8] Most C statements for pointer manipulation can be translated to these statements, including most type casts and restricted pointer arithmetic.

[9] No abstraction for such data is considered.

Heap representation As discussed in Sect. 2, we encode a single heap configuration as a deterministic $(Sel \cup Data \cup Var)$-labelled hypergraph with the ranking function being such that $\#(x) = 1 \Leftrightarrow x \in Sel$ and $\#(x) = 0 \Leftrightarrow x \in Data \cup Var$. In the hypergraph, the nodes represent allocated memory cells, unary hyperedges (labelled by symbols from *Sel*) represent selectors, and the nullary hyperedges (labelled by symbols from $Data \cup Var$) represent data values and program variables.[10] Input ports of the hypergraphs are nodes pointed to by program variables. Null and undefined values are modelled as two special nodes `null` and `undef`. We represent sets of heap configurations as hierarchical $(Sel \cup Data \cup Var)$-labelled SCFA.

Symbolic execution The symbolic computation of reachable heap configurations is done over a control flow graph (CFG) obtained from the source program. A control flow action a applied to a hypergraph G (i.e., to a single configuration) returns a hypergraph $a(G)$ that is obtained from G as follows. Non-destructive actions `x = y`, `x = y->s`, or `x = null` remove the `x`-label from its current position and label with it the node pointed by `y`, the `s`-successor of that node, or the `null` node, respectively. The destructive action `x->s = y` replaces the edge $(v_{\mathtt{x}}, \mathtt{s}, v)$ by the edge $(v_{\mathtt{x}}, \mathtt{s}, v_{\mathtt{y}})$ where $v_{\mathtt{x}}$ and $v_{\mathtt{y}}$ are the nodes pointed to by `x` and `y`, respectively. Further, `malloc(x)` moves the `x`-label to a newly created node, `free(x)` removes the node pointed to by `x` (and links `x` and all aliased variables with `undef`), and `x->data` $= \mathtt{d}_{new}$ replaces the edge $(v_{\mathtt{x}}, \mathtt{d}_{old})$ by the edge $(v_{\mathtt{x}}, \mathtt{d}_{new})$. Evaluating a guard g applied on G amounts to a simple test of equality of nodes or equality of data fields of nodes. Dereferences of `null` and `undef` are of course detected (as an attempt to follow a non-existing hyperedge) and an error is announced. Emergence of garbage is detected iff $a(G)$ is not well-connected.[11]

We, however, compute not on single hypergraphs representing particular heaps but on sets of them represented by hierarchical SCFA. For now, we assume the nested SCFA used to be provided by the user. For a given control flow action (or guard) x and a hierarchical SCFA $\mathcal{S}$, we need to symbolically compute an SCFA $x(\mathcal{S})$ s.t. $[\![x(\mathcal{S})]\!]$ equals $\{x(G) \mid G \in [\![\mathcal{S}]\!]\}$ if x is an action and $\{G \in [\![\mathcal{S}]\!] \mid x(G)\}$ if x is a guard.

Derivation of the SCFA $x(\mathcal{S})$ from $\mathcal{S}$ involves several steps. The first phase is *materialisation* where we unfold nested SFA representing boxes that hide data values or pointers referred to by x. We note that we are unfolding only SFA in the closest neighbourhood of the involved pointer variables; thus, on the level of TA, we touch only nested SFA adjacent to root-points. In the next phase, we introduce *additional root-points* for every node referred to by x to the forest representation. Third, we perform the *actual update*, which due to the previous step amounts to manipulation with root-points only [12]. Last, we repeatedly *fold (apply) boxes* and *normalise* (transform the obtained SFA into a canonicity respecting form) until no further box can be applied, so that we end up with an SCFA. We note that like the operation of unfolding, folding is also done only in the closest neighbourhood of root-points.

Unfolding is, loosely speaking, done by replacing a TA rule labelled by a nested SFA by the nested SFA itself (plus the appropriate binding of states of the top-level SFA to ports of the nested SFA). Folding is currently based on detecting isomorphism of a part of the top-level SFA and a nested SFA. The part of the top-level SFA is then replaced by a single

[10] Below, to simplify the informal description, we say that a node is labelled by a variable instead of saying that the variable labels a nullary hyperedge leaving from that node.

[11] Further, we note that we also handle a restricted pointer arithmetic. This is basically done by indexing elements of *Sel* by integers to express that the target of a pointer is an address of a memory cell plus or minus a certain offset. The formalism described in the paper may be easily adapted to support this feature.

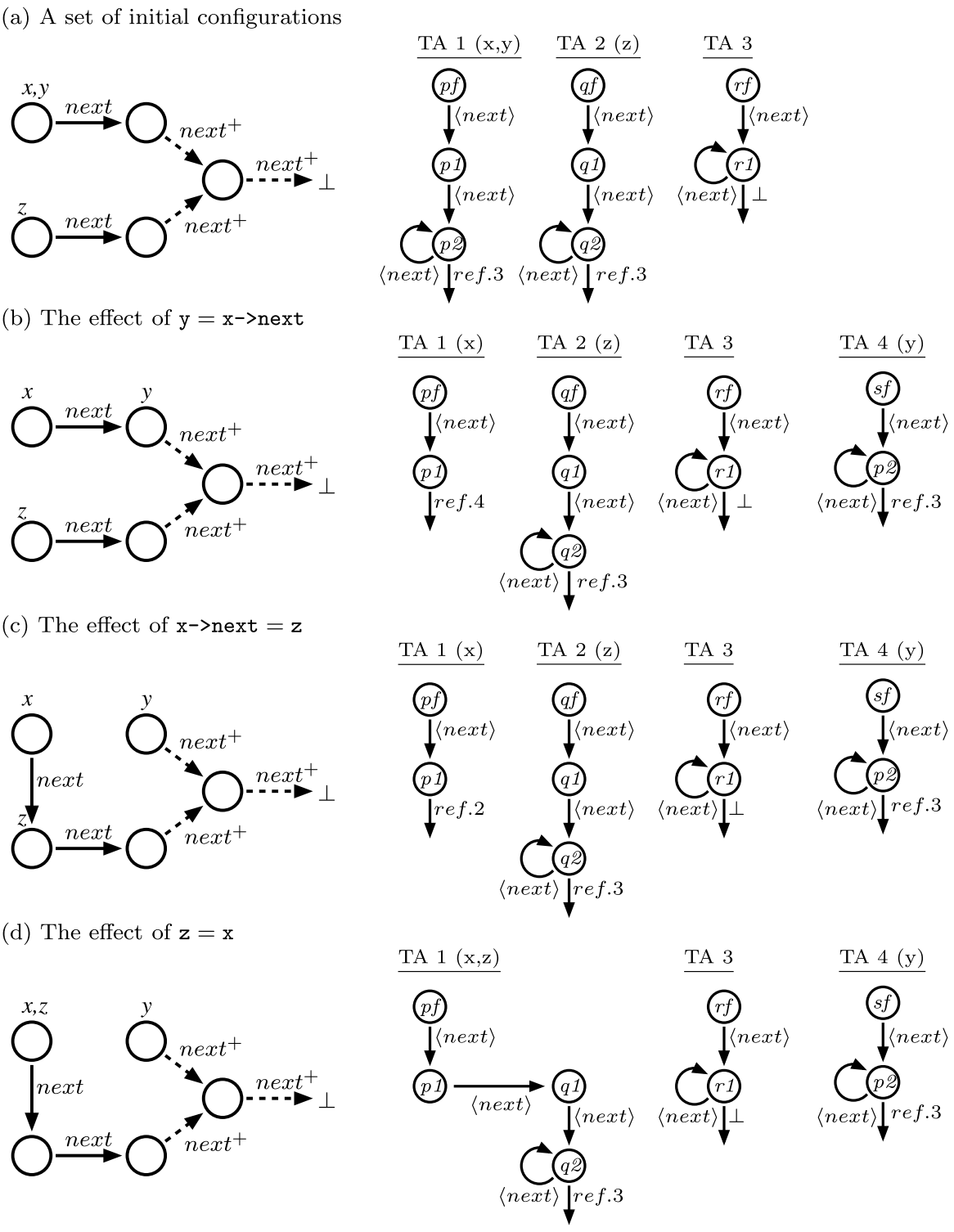


Fig. 4 A concrete (on the *left*) and symbolic execution (on the *right*) of statements `y = x->next`, `x->next = z`, and `z = x`. For the sake of simplicity, the presented FA are not strictly in their canonical form

rule labelled by the nested SFA. Note that this may be further improved by using language inclusion instead of isomorphism of automata.

A simplified example of a symbolic execution is provided in Fig. 4. In the left part of the figure, we provide concrete heaps (the dashed edges represent sequences of one or more edges linked into a linked-list), and in the right part, we provide their forest automata repre-

sentation (for a better readability, top-down tree automata are used). The initial configuration is depicted in Figs. 4(a), and (b), (c), and (d) represent the sets of heaps obtained after successively applying the statements `x = y->next`, `x->next = z`, and `z = x`.

The fixpoint computation The verification procedure performs a classical (forward) control-flow fixpoint computation over the CFG where flow values are hierarchical SCFA that represent sets of possible heap configurations at particular program locations. We start from the input location with the SCFA representing an empty heap with all variables undefined. The join operator is the union of SCFA. With every edge from a source location l labelled by x (an action or a guard), we associate the flow transfer function f_x. The function f_x takes the flow value (SCFA) $\mathcal{S}$ at l as its input and (1) computes the SCFA $x(\mathcal{S})$, (2) applies *abstraction* to $x(\mathcal{S})$, and returns the result.

The abstraction may be implemented by applying the general techniques described in the framework of abstract regular tree model checking [6] to the individual TA inside FA. Particularly, the abstraction collapses states with similar languages (based on their languages up-to certain tree depth or using predicate languages).

To detect spurious counterexamples and to refine abstraction, one can use a *backward run* similarly as in [6]. This is possible since the steps of the symbolic execution may be reversed, and it is also possible to safely approximate intersections of hierarchical SFA. More precisely, given SCFA $\mathcal{S}_1$ and $\mathcal{S}_2$, one can compute an SCFA $\mathcal{S}$ such that $[\![\mathcal{S}]\!] \subseteq [\![\mathcal{S}_1]\!] \cap [\![\mathcal{S}_2]\!]$. This under-approximation is safe since it can lead neither to false positives nor to false negatives (it can only cause the computation not to terminate). Moreover, for the SCFA that appear in the case studies in this paper, the intersection we compute is actually precise. More details can be found in [12].

6 Implementation and experimental results

We have implemented the proposed approach in a prototype tool called *Forester*, having the form of a `gcc` plug-in. The core of the tool is our own library of TA that uses the recent technology for handling nondeterministic automata (particularly, methods for reducing the size of TA and for testing language inclusion on them [2, 3]). The fixpoint computation is accelerated by the so-called finite height abstraction that is based on collapsing states of TA that have the same languages up to certain depth [6].

Although our implementation is a prototype, the results are very encouraging with regard to the generality of structures the tool can handle, precision of the generated invariants as well as the running times. We tested the tool on sample programs with various types of lists (singly-linked, doubly-linked, cyclic, nested), trees, and their combinations. Basic memory safety properties—in particular, absence of null and undefined pointer dereferences, double free operations, and absence of garbage—were checked.

We have compared the performance of our tool with that of Space Invader [4], the first fully automated tool based on separation logic, Predator [10], a new fully automated tool based in principle on separation logic (although it represents sets of heaps using graphs), and also with the ARTMC tool [7] based on abstract regular tree model checking.[12] The comparison with Space Invader and Predator was done on examples with lists only since Invader and Predator do not handle trees. The higher flexibility of our automata abstraction

[12]Since it is quite difficult to encode the input for ARTMC, we have tried it on some interesting cases only.

Table 1 Experimental results

Example	Forester	Invader	Predator	ARTMC
SLL (delete)	0.01	0.10	0.01	0.50
SLL (reverse)	<0.01	0.03	<0.01	
SLL (bubblesort)	0.02	Err	0.02	
SLL (insertsort)	0.02	0.10	0.01	
SLL (mergesort)	0.07	Err	0.13	
SLL of CSLLs	0.07	T	0.12	
SLL+head	0.01	0.06	0.01	
SLL of 0/1 SLLs	0.02	T	0.03	
SLL_{Linux}	<0.01	T	<0.01	
DLL (insert)	0.02	0.08	0.03	0.40
DLL (reverse)	0.01	0.09	0.01	1.40
DLL (insertsort1)	0.20	0.18	0.15	1.40
DLL (insertsort2)	0.06	Err	0.03	
CDLL	<0.01	0.09	<0.01	
DLL of CDLLs	0.18	T	0.13	
SLL of $2CDDLs_{Linux}$	0.03	T	0.19	
tree	0.06			3.00
tree+stack	0.02			
tree+parents	0.10			
tree (DSW)	0.16			o.o.m

shows up, for example, in the test case with a list of sublists of lengths 0 or 1 (discussed already in the introduction) for which Space Invader does not terminate. Our technique handles this example smoothly (without any need to add special inductive predicates that could decrease the performance or generate false alarms). Predator can also handle this test case, but to achieve that, the algorithms implemented in it must have been manually extended to use a new kind of list segment of length 0 or 1, together with an appropriate modification of the implementation of Predator's join and abstraction operations.[13] On the other hand, the ARTMC tool can, in principle, handle more general structures than we can currently handle such as trees with linked leaves. However, the used representation of heap configurations is much heavier which causes ARTMC not to scale that well.

Table 1 summarises running times (in seconds) of the four tools on our case studies. The value T means that the running time exceeded 30 minutes, o.o.m. means that the tool ran out of memory, and the value Err stands for a failure of symbolic execution. The names of experiments in the table contain the name of the data structure handled by the program. In particular, "SLL" stands for singly-linked lists, "DLL" for doubly linked lists (the prefix "C" means cyclic), "tree" for binary trees, "tree+parents" for trees with parent pointers. Nested variants of SLL are named as "SLL of" and the type of the nested list. In particular, "SLL of 0/1 SLLs" stands for SLL of nested SLL of length 0 or 1. "SLL+head" stands for a list where each element points to the head of the list, "SLL of 2CDLLs" stands for SLL whose implementation of lists used in the Linux kernel with restricted pointer arithmetic [10] which

[13]The operations were carefully tuned not to easily generate false alarms, but the risk of generating them has anyway been increased.

we can also handle. All experiments start with a random creation and end with a disposal of the specified structure. If some further operation is performed in between the creation phase and the disposal phase, it is indicated in brackets. In the experiment "tree+stack", a randomly created tree is disposed using a stack in a top-down manner such that we always dispose a root of a subtree and save its subtrees into the stack. "DSW" stands for the Deutsch-Schorr-Waite tree traversal (the Lindstrom variant). We have run our tests on a machine with Intel T9600 (2.8 GHz) CPU and 4 GB of RAM.

7 Conclusion

We have proposed hierarchically nested forest automata as a new means of encoding sets of heap configurations when verifying programs with dynamic linked data structures. The proposal brings the principle of separation from separation logic into automata, allowing us to combine some advantages of automata (generality, less rigid abstraction) with a better scalability stemming from local heap manipulation. We have shown some interesting properties of our representation from the point of view of inclusion checking. We have implemented and tested the approach on multiple non-trivial cases studies, demonstrating the approach to be promising.

In the future, we plan to improve the implementation of our tool Forester, including a support for predicate language abstraction within abstract regular tree model checking [6]. We also plan to implement the automatic learning of nested FA. From a more theoretical perspective, it is interesting to show whether inclusion checking is or is not decidable for the full class of nested FA. Another interesting direction is then a possibility of allowing truly recursive nesting of FA, which would allow us to handle very general structures such as trees with linked leaves.

References

1. Abdulla PA, Bouajjani A, Cederberg J, Haziza F, Rezine A (2008) Monotonic abstraction for programs with dynamic memory heaps. In: Proc of CAV'08. LNCS, vol 5123. Springer, Berlin
2. Abdulla PA, Bouajjani A, Holík L, Kaati L, Vojnar T (2008) Computing simulations over tree automata: efficient techniques for reducing TA. In: Proc of TACAS'08. LNCS, vol 4963
3. Abdulla PA, Chen Y-F, Holík L, Mayr R, Vojnar T (2010) When simulation meets antichains (on checking language inclusion of NFAs). In: Proc of TACAS'10. LNCS, vol 6015. Springer, Berlin
4. Berdine J, Calcagno C, Cook B, Distefano D, O'Hearn PW, Wies T, Yang H (2007) Shape analysis for composite data structures. In: Proc CAV'07. LNCS, vol 4590. Springer, Berlin
5. Bouajjani A, Bozga M, Habermehl P, Iosif R, Moro P, Vojnar T (2006) Programs with lists are counter automata. In: Proc of CAV'06. LNCS, vol 4144. Springer, Berlin
6. Bouajjani A, Habermehl P, Rogalewicz A, Vojnar T (2006) Abstract regular tree model checking. Electron Notes Theor Comput Sci 149(1):37–48
7. Bouajjani A, Habermehl P, Rogalewicz A, Vojnar T (2006) Abstract regular tree model checking of complex dynamic data structures. In: Proc of SAS'06. LNCS, vol 4134. Springer, Berlin
8. Calcagno C, Distefano D, O'Hearn PW, Yang H (2009) Compositional shape analysis by means of Bi-abduction. In: Proc of POPL'09. ACM, New York
9. Deshmukh JV, Emerson EA, Gupta P (2006) Automatic verification of parameterized data structures. In: Proc of TACAS'06. LNCS, vol 3920. Springer, Berlin
10. Dudka K, Peringer P, Vojnar T (2011) Predator: a practical tool for checking manipulation of dynamic data structures using separation logic. In: Proc of CAV'11. LNCS, vol 6806. Springer, Berlin
11. Guo B, Vachharajani N, August DI (2007) Shape analysis with inductive recursion synthesis. In: Proc of PLDI'07. ACM, New York
12. Habermehl P, Holík L, Rogalewicz A, Šimáček J, Vojnar T (2011) Forest automata for verification of heap manipulation. Technical report FIT-TR-2011-01, FIT BUT, Czech Republic. http://www.fit.vutbr.cz/~isimacek/pub/FIT-TR-2011-01.pdf

13. Madhusudan P, Parlato G, Qiu X (2011) Decidable logics combining heap structures and data. In: Proc of POPL'11. ACM, New York
14. Møller A, Schwartzbach M (2001) The pointer assertion logic engine. In: Proc of PLDI'01. ACM, New York
15. Nguyen HH, David C, Qin S, Chin WN (2007) Automated verification of shape and size properties via separation logic. In: Proc of VMCAI'07. LNCS, vol 4349. Springer, Berlin
16. Reynolds JC (2002) Separation logic: a logic for shared mutable data structures. In: Proc of LICS'02. IEEE Comput Soc, Los Alamitos
17. Sagiv S, Reps TW, Wilhelm R (2002) Parametric shape analysis via 3-valued logic. ACM Trans Program Lang Syst 24(3):217–298
18. Yang H, Lee O, Calcagno C, Distefano D, O'Hearn PW (2007) On scalable shape analysis. Technical report RR-07-10, Queen Mary, University of London
19. Yang H, Lee O, Berdine J, Calcagno C, Cook B, Distefano D, O'Hearn PW (2008) Scalable shape analysis for systems code. In: Proc of CAV'08. LNCS, vol 5123. Springer, Berlin
20. Zee K, Kuncak V, Rinard M (2008) Full functional verification of linked data structures. In: Proc of PLDI'08. ACM, New York

String Constraints for Verification*

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], Lukáš Holík[3], Ahmed Rezine[4], Philipp Rümmer[1], and Jari Stenman[1]

[1] Department of Information Technology, Uppsala University, Sweden
[2] Institute of Information Science, Academia Sinica, Taiwan
[3] Faculty of Information Technology, Brno University of Technology, Czech Republic
[4] Department of Computer and Information Science, Linköping University, Sweden

Abstract. We present a decision procedure for a logic that combines (i) word equations over string variables denoting words of arbitrary lengths, together with (ii) constraints on the length of words, and on (iii) the regular languages to which words belong. Decidability of this general logic is still open. Our procedure is sound for the general logic, and a decision procedure for a particularly rich fragment that restricts the form in which word equations are written. In contrast to many existing procedures, our method does not make assumptions about the maximum length of words. We have developed a prototypical implementation of our decision procedure, and integrated it into a CEGAR-based model checker for the analysis of programs encoded as Horn clauses. Our tool is able to automatically establish the correctness of several programs that are beyond the reach of existing methods.

1 Introduction

Software model checking is an active research area that has witnessed a remarkable success in the past decades [15,8]. Model checking tools are already used in industrial applications [2]. One reason for this success is recent developments in SMT technology [5,7,3], which allow efficient symbolic representations of different data types in programs. This dependence encompasses, however, that model checking tools are inherently limited by the data types that can be handled by the underlying SMT solver. A data type for which satisfying decision procedures have been missing is that of *strings*. Our work proposes a rich string logic together with a decision procedure targeting model checking applications.

String data types are present in programming and scripting languages. In fact, it is impossible to capture the essence of many programs, for instance in database and web applications, without the ability to precisely represent and reason about string data types. The control flow of programs can depend on words denoted by string variables, on the length of words, or on regular languages to which they belong. For example, a program allowing users to choose a username and

* Supported by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the Czech Science Foundation (13-37876P), Brno University of Technology (FIT-S-12-1, FIT-S-14-2486), and the Linköping CENIIT Center (12.04).

A. Biere and R. Bloem (Eds.): CAV 2014, LNCS 8559, pp. 150–166, 2014.

a password may require the password to be of a minimal length, to be different from the username, and to be free from invalid characters. Reasoning about such constraints is also crucial when verifying that database and web applications are free from SQL injections and other security vulnerabilities.

Existing solvers for programs manipulating string variables and their length are either unsound, not expressive enough, or lack the ability to provide counterexamples. Many solvers [9,23,24] are unsound since they assume an a priori fixed upper bound on the length of the possible words. Others [9,17,26] are not expressive enough as they do not handle word equations, length constraints, or membership predicates. Such solvers are mostly aimed at performing symbolic executions, i.e., establishing feasibility of paths in a program. The solver in [25] performs sound over-approximation, but without supplying counterexamples in case the verification fails. In contrast, our decision procedure specifically targets model checking applications. In fact, we use it in a prototype model checker in order to automatically establish program correctness for several examples.

Our decision procedure establishes satisfiability of formulae written as Boolean combinations of: (i) word (dis)equalities such as $(a \cdot u = v \cdot b)$ or $(a \cdot u \neq v \cdot b)$, where a, b are letters and u, v are string variables denoting words of arbitrary lengths, (ii) length constraints such as $(|u| = |v| + 1)$, where $|u|$ refers to the length of the word denoted by string variable u, and (iii) predicates representing membership in regular expressions, e.g., $u \in c \cdot (a + b)^*$. Each of these predicates can be crucial for capturing the behavior and establishing the correctness of a string-manipulating program (cf. the program in Section 2). The analysis is not trivial as it needs to capture subtle interactions between different types of predicates. For instance, the formulae $\phi_1 = (a \cdot u = v \cdot b) \wedge (|u| = |v| + 1)$ and $\phi_2 = (a \cdot u = v \cdot b) \wedge v \in c \cdot (a + b)^*$ are unsatisfiable, i.e., there is no possible assignment of words to u and v that makes the conjunctions evaluate to true. The analysis then needs to propagate facts from one type of predicates to another; e.g., in ϕ_1 the analysis deduces from $(a \cdot u = v \cdot b)$ that $|u| = |v|$, which results in an unsatisfiable formula $(|u| = |v| \wedge |u| = |v| + 1))$. The general decidability problem is still open. We guarantee termination of our procedure for a fragment of the logic including the three types of predicates. The fragment we consider is rich enough to capture all the practical examples we have encountered.

We have integrated our decision procedure in a prototype model checker and used it to verify properties of implementations of common string manipulating functions such as the Hamming and Levenshtein distances. Predicates required for verification can be provided by hand; to achieve automation, in addition we propose a constraint-based interpolation procedure for regular word constraints. In combination with our decision procedure for words, this enables us to automatically analyze programs that are currently beyond the reach of state-of-the-art software model checkers.

Related Work. The pioneering work by Makanin [18] proposed a decision procedure for word equations (i.e., Boolean combinations of (dis)equalities) where the variables can denote words of arbitrary lengths. The decidability problem is already open [4] when word equations are combined with length constraints of

the form $|u| = |v|$. Our logic adds predicates representing membership in regular languages to word equations and length constraints. This means that decidability is still an open problem. A contribution of our work is the definition of a rich sub-logic for which we guarantee the termination of our procedure.

In a work close to ours, the authors in [10] show decidability of a logic that is strictly weaker than the one for which we guarantee termination. For instance, in [10], membership predicates are allowed only under the assumption that no string variables can appear in the right hand sides of the equality predicates. This severely restricts the expressiveness of the logic. In [26], the authors augment the Z3 [7] SMT solver in order to handle word equations with length constraints. However, they do not support regular membership predicates. In our experience, these are crucial during model checking based verification.

Finally, in addition to considering more general equations, our work comes with an interpolation-based verification technique adapted for string programs. Notice that neither of [10,26] can establish correctness of programs with loops.

Outline. In the next section, we use a simple program to illustrate our approach. In Section 3 we introduce a logic for word equations with arithmetic and regular constraints, and then describe in Section 4 a procedure for deciding satisfiability of formulae in the logic. In Section 5 we define a class formulae for which we guarantee the termination of our decision procedure. We describe the verification procedure in Section 6 and the implementation effort in Section 7. Finally in Section 8 we give some conclusions and directions for future work.

2 A Simple Example

In this section, we use the simple program listed in Fig. 1 to give a flavor of our verification approach. The listing makes use of features that are common in string manipulating programs. We will argue that establishing correctness for such programs requires: (i) the ability to refer to string variables of arbitrary lengths, (ii) the ability to express combinations of constraints, like that the words denoted by the variables belong to regular expressions, that their lengths obey arithmetic inequalities, or that the words themselves are solutions to word equations, and (iii) the ability for a decision procedure to precisely capture the subtle interaction between the different kinds of involved constraints.

In the program of Fig. 1, a string variable `s` is initialized with the empty word. A loop is then executed an arbitrary number of times. At each iteration of the loop, the instruction `s= 'a' + s + 'b'` appends the letter `'a'` at the beginning of variable `s` and the letter `'b'` at its end. After the loop, the program asserts that `s` does not have the word `'ba'` as a substring (denoted by `!s.contains('ba')`, and that its length (denoted by `s.length()`) is even.

Observe that the string variable `s` does not assume a maximal length. Any verification procedure that requires an a priori fixed bound on the length of the string variables is necessarily unsound and will fail to establish correctness.

Moreover, establishing correctness requires the ability to express and to reason about predicates such as those mentioned in the comments of the code in Fig. 1.

```
// Pre = (true)
String s= '';
// P1 = (s ∈ ϵ)
while(*){
    // P2 = (s = u · v ∧ u ∈ a* ∧ v ∈ b* ∧ |u| = |v|)
    s= 'a' + s + 'b';
}
// P3 = P2
assert(!s.contains('ba') && (s.length() % 2) == 0);
// Post = P3
```

Fig. 1. A simple program manipulating a string variable s. Our logic allows to precisely capture the word equations, membership predicates and length constraints that are required for validating the assertion is never violated. Our decision procedure can then automatically validate the required verification conditions described in Fig. 2.

$$vc_1 : post(Pre, \texttt{s} = \text{""}) \implies P_1 \qquad vc_2 : P_1 \implies P_2$$
$$vc_3 : post(P_2, \texttt{s} = \text{"a"} \cdot \texttt{s} \cdot \text{"b"}) \implies P_2 \qquad vc_4 : P_2 \implies P_3$$
$$vc_5 : post(P_3, \texttt{assume(s.contains("ba") || !(s.length()\%2 ==0))}) \implies false$$
$$vc_6 : post(P_3, \texttt{assume(!s.contains("ba") \&\& (s.length()\%2 ==0))}) \implies Post$$

Fig. 2. Verification conditions for the simple program of Fig. 1

For instance, the loop invariant P_2 states that: (i) the variable s denotes a finite word w_s of arbitrary length, (ii) that w_s equals the concatenation of two words w_u and w_v, (iii) that $w_u \in a^*$ and $w_v \in b^*$, and (iv) that the length $|w_u|$ of word w_u equals the length $|w_v|$ of word w_v.

Using the predicates in Fig. 1, we can formulate program correctness in terms of the validity of each of the implications listed in Fig. 2. For instance, validity of the verification condition vc_5 amounts to showing that $\neg vc_5 = (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v|) \wedge (s = s_1 \cdot b \cdot a \cdot s_2 \vee \neg(|s| = 2n))$ is unsatisfiable. To establish this result, our decision procedure generates the two proof obligations $\neg vc_{51} : (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v| \wedge s = s_1 \cdot b \cdot a \cdot s_2)$ and $\neg vc_{52} : (s = u \cdot v \wedge u \in a^* \wedge v \in b^* \wedge |u| = |v| \wedge \neg(|s| = 2n))$.

In order to check vc_{51}, the procedure symbolically matches all the possible ways in which a word denoted by $u \cdot v$ can also be denoted by $s_1 \cdot b \cdot a \cdot s_2$. For instance, $u = s_1 \cdot b \wedge v = a \cdot s_2$ is one possible matching. In order to be able to show unsatisfiability, the decision procedure has to also consider the other possible matchings. For instance, the case where the word denoted by u is a strict prefix of the one denoted by s_1 has also to be considered. For this reason, the matching process might trigger new matchings. In general, there is no guarantee that the sequence of generated matchings will terminate. However, we show that this sequence terminates for an expressive fragment of the logic. This fragment includes the predicates of mentioned in this section and all predicates we encountered in practical programs, The procedure then checks satisfiability of each such a matching. For instance, the matching $u = s_1 \cdot b \wedge v = a \cdot s_2$

is shown to be unsatisfiable due the the membership predicate $v \in b^*$. In fact our procedure automatically proves that $\neg v_{51}$ is not satisfiable after checking all possible matchings.

So for $\neg vc_5$ to be satisfiable, $\neg vc_{52}$ needs to be satisfiable. Our procedure deduces that this would imply that $|u| = |v| \wedge \neg(|u| + |v| = 2n)$ is satisfiable. We leverage on existing standard decision procedures for linear arithmetic in order to show that this is not the case. Hence $\neg vc_5$ is unsatisfiable and vc_5 is valid. For this example, and those we report on in Section 6, our procedure can establish correctness fully automatically given the required predicates.

Observe that establishing validity requires the ability to capture interactions among the different types of predicates. For instance, establishing validity of vc_5 involves the ability to combine the word equations $(s = u \cdot v \wedge s = s_1 \cdot b \cdot a \cdot s_2)$ with the membership predicates $(u \in a^* \wedge v \in b^*)$ for vc_{51}, and with the length constraints $(|u| = |v| \wedge \neg(|s| = 2n))$ for vc_{52}. Capturing such interactions is crucial for establishing correctness and for eliminating false positives.

3 Defining the String Logic $\mathcal{E}_{e,r,l}$

In this section we introduce a logic, which we call $\mathcal{E}_{e,r,l}$, for word equations, regular constraints (short for membership constraints in regular languages) and length and arithmetic inequalities. We assume a finite alphabet Σ and write Σ^* to mean the set of finite words over Σ. We work with a set U of string variables denoting words in Σ^* and write $\mathcal{Z}$ for the set of integer numbers.

Syntax. We let variables u, v range over the set U. We write $|u|$ to mean the length of the word denoted by variable u, k to mean an integer in $\mathcal{Z}$, c to mean a letter in Σ and w to mean a word in Σ^*. The syntax of formulae in $\mathcal{E}_{e,r,l}$ is defined as follows:

$$
\begin{array}{rcll}
\phi & ::= & \phi \wedge \phi \mid \neg\phi \mid \varphi_e \mid \varphi_l \mid \varphi_r & \text{formulae} \\
\varphi_e & ::= & tr = tr \mid tr \neq tr & \text{(dis)equalities} \\
\varphi_l & ::= & e \leq e & \text{arithmetic inequalities} \\
\varphi_r & ::= & tr \in \mathcal{R} & \text{membership predicates} \\
tr & ::= & \epsilon \mid c \mid u \mid tr \cdot tr & \text{terms} \\
\mathcal{R} & ::= & \emptyset \mid \epsilon \mid c \mid w \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} + \mathcal{R} \mid \mathcal{R} \cap \mathcal{R} \mid \mathcal{R}^C \mid \mathcal{R}^* & \text{regular expressions} \\
e & ::= & k \mid |tr| \mid k * e \mid e + e & \text{integer expressions}
\end{array}
$$

Assume variables $\{u_i\}_{i=1}^n$, terms $\{tr_i\}_{i=1}^n$ and integer expressions $\{e_i\}_{i=1}^n$. We write $\phi[u_1/tr_1] \ldots [u_n/tr_n]$ (resp. $\phi[|u_1|/e_1] \ldots [|u_n|/e_n]$) to mean the formula obtained by syntactically substituting in ϕ each occurrence of u_i by term tr_i (resp. each occurrence of $|u_i|$ by expression e_i). Such a substitution is said to be well-defined if no variable u_i (resp. $|u_i|$) appears in any tr_i (resp. e_i).

The set of word variables appearing in a term is defined as follows: $Vars(\epsilon) = \emptyset$, $Vars(c) = \emptyset$, $Vars(u) = \{u\}$ and $Vars(tr_1 \cdot tr_2) = Vars(tr_1) \cup Vars(tr_2)$.

Semantics. The semantics of $\mathcal{E}_{e,r,l}$ is mostly standard. We describe it using a mapping η (called *interpretation*) that assigns words in Σ^* to string variables in U. We extend η to terms as follows: $\eta(\epsilon) = \epsilon$, $\eta(c) = c$ and $\eta(tr_1.tr_2) = \eta(tr_1).\eta(tr_2)$. Every regular expression $\mathcal{R}$ is evaluated to the language $\mathcal{L}(\mathcal{R})$ it represents. Given an interpretation η, we define another mapping β_η that associates a number in $\mathcal{Z}$ to integer expressions as follows: $\beta_\eta(k) = k$, $\beta_\eta(|u|) = |\eta(u)|$, $\beta_\eta(|tr|) = |\eta(tr)|$, $\beta_\eta(k * e) = k * \beta_\eta(e)$, and $\beta_\eta(e_1 + e_2) = \beta_\eta(e_1) + \beta(e_2)$. A formula in $\mathcal{E}_{e,r,l}$ is then evaluated to a value in $\{ff, tt\}$ as follows:

$$\begin{aligned}
val_\eta(\phi_1 \wedge \phi_2) &= tt \quad \text{iff} \quad val_\eta(\phi_1) = tt \text{ and } val_\eta(\phi_2) = tt \\
val_\eta(\neg\phi_1) &= tt \quad \text{iff} \quad val_\eta(\phi_1) = ff \\
val_\eta(tr \in \mathcal{R}) &= tt \quad \text{iff} \quad \eta(tr) \in \mathcal{L}(\mathcal{R}) \\
val_\eta(tr_1 = tr_2) &= tt \quad \text{iff} \quad \eta(tr_1) = \eta(tr_2) \\
val_\eta(tr_1 \neq tr_2) &= tt \quad \text{iff} \quad \neg(\eta(tr_1) = \eta(tr_2)) \\
val_\eta(e_1 \leq e_2) &= tt \quad \text{iff} \quad \beta_\eta(e_1) \leq \beta_\eta(e_2)
\end{aligned}$$

A formula ϕ is said to be *satisfiable* if there is an interpretation η such that $val_\eta(\phi) = tt$. It is said to be *unsatisfiable* otherwise.

4 Inference Rules

In this section, we describe our set of inference rules for checking the satisfiability of formulae in the logic $\mathcal{E}_{e,r,l}$ of Section 3. Given a formula ϕ, we build a proof tree rooted at ϕ by repeatedly applying the inference rules introduced in this Section. We can assume, without loss of generality, that the formula is given in Disjunctive Normal Form. An inference rule is of the form:

$$\textsc{Name} : \frac{B_1 \; B_2 \; ... \; B_n}{A} \; cond$$

In this inference rule, NAME is the name of the rule, *cond* is a side condition on A for the application of the rule, $B_1 \; B_2 \; ... \; B_n$ are called premises, and A is called the conclusion of the rule. (We omit the side condition *cond* from NAME when it is tt.) The premises and conclusion are formulae in $\mathcal{E}_{e,r,l}$. Each application consumes a conclusion and produces the set of premises. The inference rule is said to be *sound* if the satisfiability of the conclusion implies the satisfiability of one of the premises. It is said to be *locally complete* if the satisfiability of one of the premises implies the satisfiability of the conclusion. If all inference rules are locally complete, and if ϕ or one of the produced premises turns out to be satisfiable, then ϕ is also satisfiable. If all the inference rules are sound and none of the produced premises is satisfiable, then ϕ is also unsatisfiable.

We organize the inference rules in four groups. We use the rules of the first group to eliminate disequalities. The rules of the second group are used to simplify equalities. The rules of the third group are used to eliminate membership predicates. The rules of the last group are used to propagate length constraints. In addition, we assume standard decision procedures [3] for integer arithmetic.

Lemma 1. *The inference rules of this section are sound and locally complete.*

4.1 Removing Disequalities

We use rules Not-Eq and Diseq-Split in order to eliminate disequalities. In rule Not-Eq, we establish that $tr \neq tr \wedge \phi$ is not satisfiable and close this branch of the proof. In the second rule Diseq-Split, we eliminate disequalities involving arbitrary terms. For this, we make use of the fact that the alphabet Σ is finite and replace any disequality with a finite set of equalities. More precisely, assume a formula $tr \neq tr' \wedge \phi$ in $\mathcal{E}_{e,r,l}$. We observe that the disequality $tr \neq tr'$ holds iff the words w_{tr} and $w_{tr'}$ denoted by the terms tr and tr' are different. This corresponds to one of three cases. Assume three fresh variables u, v and v'. In the first case, the words w_{tr} and $w_{tr'}$ contain different letters $c \neq c'$ after a common prefix w_u. They are written as the concatenations $w_u \cdot c \cdot w_v$ and $w_u \cdot c' \cdot w_{v'}$ respectively. We capture this case using the set $\textsc{Split}_{\textsc{Diseq-Split}} = \{tr = u \cdot c \cdot v \wedge tr' = u \cdot c' \cdot v' \wedge \phi \mid c, c' \in \Sigma \text{ and } c \neq c'\}$. In the second case, the word $w_{tr'} = w_u$ is a strict prefix of $w_{tr} = w_u \cdot c \cdot w_v$. We capture this with $\textsc{Split}'_{\textsc{Diseq-Split}} = \{tr = u \cdot c \cdot v \wedge tr' = u \wedge \phi \mid c \in \Sigma\}$. In the third case, the word $w_{tr} = w_u$ is a strict prefix of $w_{tr'} = w_u \cdot c' \cdot w'_v$, and we capture this case using the set $\textsc{Split}''_{\textsc{Diseq-Split}} = \{tr = u \wedge tr' = u \cdot c \cdot v' \wedge \phi \mid c \in \Sigma\}$.

$$\textsc{Not-Eq} : \frac{*}{tr \neq tr \wedge \phi} \qquad \textsc{Eq} : \frac{\phi}{tr = tr \wedge \phi}$$

$$\textsc{Diseq-Split} : \frac{\textsc{Split}_{\textsc{Diseq-Split}} \cup \textsc{Split}'_{\textsc{Diseq-Split}} \cup \textsc{Split}''_{\textsc{Diseq-Split}}}{tr \neq tr' \wedge \phi}$$

4.2 Simplifying Equalities

We introduce rules Eq, Eq-Var, and Eq-Word to manipulate equalities. Rule applications take into account symmetry of the equality operator (i.e., if a rule can apply to $w \cdot tr_1 = tr_2 \wedge \phi$ then it can also apply to $tr_2 = w \cdot tr_1 \wedge \phi$). Rule Eq eliminates trivial equalities of the form $tr = tr$.

Rule Eq-Var eliminates variable u from the equality $u \cdot tr_1 = tr_2 \wedge \phi$. Let w_u be some word denoted by u. For the equality to hold, w_u must be a prefix of the word denoted by tr_2. There are two cases. The first case, represented by $\textsc{Split}_{\textsc{Eq-Var}}$ in Eq-Var, captures situations where w_u coincides with a word denoted by a prefix tr_3 of tr_2. The second case, represented by $\textsc{Split}'_{\textsc{Eq-Var}}$, captures situations where w_u does not coincide with a word denoted by a prefix of tr_2. Instead, tr_2 can be written as $tr_3 \cdot v \cdot tr_4$ and the word w_u is written as the concatenation of two words, one that is denoted by tr_3 and another that is prefix of the word denoted by v.

$$\textsc{Eq-Var} : \frac{\textsc{Split}_{\textsc{Eq-Var}} \cup \textsc{Split}'_{\textsc{Eq-Var}}}{u \cdot tr_1 = tr_2 \wedge \phi}$$

The set $\textsc{Split}_{\textsc{Eq-Var}}$ captures the first case, when w_u coincides with a word denoted by a prefix tr_3 of tr_2. The premises for this case are partitioned into two sets, namely $\textsc{Split}_{\textsc{Eq-Var-1}}$ and $\textsc{Split}_{\textsc{Eq-Var-2}}$:

$$\textsc{Split}_{\text{Eq-Var-1}} = \left\{ \begin{array}{l} (tr_1 = tr_4 \wedge \phi)[u/tr_3] \mid \\ tr_2 = tr_3 \cdot tr_4 \text{ and } u \text{ does not syntactically appear in } tr_3 \end{array} \right\}$$

$$\textsc{Split}_{\text{Eq-Var-2}} = \left\{ \begin{array}{l} tr_1 = tr_4 \wedge tr_5 \cdot tr_6 \in \epsilon \wedge \phi \mid \\ tr_2 = tr_3 \cdot tr_4 \text{ and } tr_3 = tr_5 \cdot u \cdot tr_6 \end{array} \right\}$$

Variable u is eliminated from the premises contained in the set $\textsc{Split}_{\text{Eq-Var-1}}$. The second set $\textsc{Split}_{\text{Eq-Var-2}}$ captures cases where u does syntactically appear in tr_3. Variable u might still appear in some of the premises of $\textsc{Split}_{\text{Eq-Var-2}}$.

The set $\textsc{Split}'_{\text{Eq-Var}}$ in Eq-Var captures the second case, namely when w_u does not coincide with a word denoted by a prefix of tr_2, written $tr_3 \cdot v \cdot tr_4$ for some variable v. The premises in $\textsc{Split}'_{\text{Eq-Var}}$ are partitioned into two sets, namely $\textsc{Split}'_{\text{Eq-Var-1}}$ and $\textsc{Split}'_{\text{Eq-Var-2}}$:

$$\textsc{Split}'_{\text{Eq-Var-1}} = \left\{ \begin{array}{l} \big((tr_1 = v_2 \cdot tr_4 \wedge \phi)[u/tr_3 \cdot v_1]\big)[v/v_1 \cdot v_2] \mid \\ tr_2 = tr_3 \cdot v \cdot tr_4 \text{ and } u \text{ appears neither in } tr_3 \text{ nor in } v \end{array} \right\}$$

$$\textsc{Split}'_{\text{Eq-Var-2}} = \left\{ \begin{array}{l} \big(tr_1 = u_2 \cdot tr_4 \wedge u_1 \cdot u_2 = tr_3 \cdot u_1 \wedge \phi\big)[u/tr_3 \cdot u_1] \mid \\ tr_2 = tr_3 \cdot u \cdot tr_4 \text{ and } u \text{ does not appear in } tr_3 \end{array} \right\}$$

The premises in $\textsc{Split}'_{\text{Eq-Var-1}}$ mention neither u nor v. The set $\textsc{Split}'_{\text{Eq-Var-2}}$ captures cases where u in the left-hand side overlaps with its occurrence on the right-hand side. Cases where u appears in tr_3 are captured in $\textsc{Split}_{\text{Eq-Var}}$.

Rule Eq-Word eliminates the word w from the equality $w \cdot tr_1 = tr_2 \wedge \phi$:

$$\textsc{Eq-Word} : \frac{\textsc{Split}_{\text{Eq-Word}} \cup \textsc{Split}'_{\text{Eq-Word}}}{w \cdot tr_1 = tr_2 \wedge \phi}$$

Again, we define two sets representing the premises of the rule:

$$\textsc{Split}_{\text{Eq-Word}} = \big\{ tr_3 \in w \wedge tr_4 = tr_1 \wedge \phi \mid tr_2 = tr_3 \cdot tr_4 \big\}$$

$$\textsc{Split}'_{\text{Eq-Word}} = \big\{ \big(tr_3 \cdot v_1 \in w \wedge v_2 \cdot tr_4 = tr_1 \wedge \phi\big)[v/v_1 \cdot v_2] \mid tr_2 = tr_3 \cdot v \cdot tr_4 \big\}$$

To simplify the presentation, we do not present suffix versions for rules Eq-Var and Eq-Word. Such rules match suffixes instead of prefixes and simply mirror the rules described above.

4.3 Removing Membership Predicates

We use rules Reg-Neg, Memb, Not-Memb, Reg-Split and Reg-Len to simplify and eliminate membership predicates. We describe them below.

Rule Reg-Neg replaces the negation of a membership predicate in a regular expression $\mathcal{R}$ with a membership predicate in its complement $\mathcal{R}^C$.

$$\textsc{Reg-Neg} : \frac{tr \in \mathcal{R}^C \wedge \phi}{\neg(tr \in \mathcal{R}) \wedge \phi}$$

Rule Memb eliminates the predicate $w \in \mathcal{R}$ in case the word w belongs to the language $\mathcal{L}(\mathcal{R})$ of the regular expression $\mathcal{R}$. If w does not belong to $\mathcal{L}(\mathcal{R})$ then rule Not-Memb closes this branch of the proof.

$$\textsc{Memb} : \frac{\phi}{w \in \mathcal{R} \wedge \phi}\ w \in \mathcal{L}(\mathcal{R}) \qquad \textsc{Not-Memb} : \frac{*}{w \in \mathcal{R} \wedge \phi}\ w \notin \mathcal{L}(\mathcal{R})$$

Rule Reg-Split simplifies membership predicates of the form $tr \cdot tr' \in \mathcal{R}$. Given such a predicate, the rule replaces it with a disjunction $\bigvee_{i=1}^{n} \left(tr \in \mathcal{R}_i \wedge tr' \in \mathcal{R}'_i\right)$ where the set $\{(\mathcal{R}_i, \mathcal{R}'_i)\}_{i=1}^{n}$ is finite and only depends on the regular expression $\mathcal{R}$. To define this set, represent $\mathcal{L}(\mathcal{R})$ using some arbitrary but fixed finite automaton (S, s_0, δ, F). Assume $S = \{s_0, \ldots, s_n\}$. Choose the regular expressions $\mathcal{R}_i, \mathcal{R}'_i$ such that : (1) $\mathcal{R}_i$ has the same language as the automaton $(S, s_0, \delta, \{s_i\})$, and (2) $\mathcal{R}'_i$ has the same language as the automaton(S, s_i, δ, F). For any word $w_{tr} \cdot w_{tr'}$ denoted by $tr \cdot tr'$ and accepted by $\mathcal{R}$, there will be a state s_i in S such that w_{tr} is accepted by $\mathcal{R}_i$ and $w_{tr'}$ is accepted by $\mathcal{R}'_i$. Given a regular expression $\mathcal{R}$, we let $\mathcal{F}(\mathcal{R})$ denote the set $\{(\mathcal{R}_i, \mathcal{R}'_i)\}_{i=1}^{n}$ above.

$$\textsc{Reg-Split} : \frac{\{tr \in \mathcal{R}' \wedge tr' \in \mathcal{R}'' \wedge \phi \mid (\mathcal{R}', \mathcal{R}'') \in \mathcal{F}(\mathcal{R})\}}{tr \cdot tr' \in \mathcal{R} \wedge \phi}$$

Rule Reg-Len can only be applied in certain cases. To identify these cases, we define the condition $\Gamma(\phi, u)$ which states, given a formula ϕ and a variable u, that u is not used in any membership predicate or in any (dis)equation in ϕ. In other words, the condition states that if u occurs in ϕ then it occurs in a length predicate. The rule Reg-Len replaces, in one step, all the membership predicates $\{u \in \mathcal{R}_i\}_{i=1}^{n}$ with an arithmetic constraint $Len(\mathcal{R}_1 \cap \ldots \cap \mathcal{R}_m, u)$. This arithmetic constraint expresses that the length $|u|$ of variable u belongs to the semi-linear set corresponding to the Parikh image of the intersection of all regular expressions $\{\mathcal{R}_i\}_{i=1}^{n}$ appearing in membership predicates of variable u. It is possible to determine a representation of this semi linear set by starting from a finite state automaton representing the intersection $\cap_i \mathcal{R}_i$ and replacing all letters with a unique arbitrary letter. The obtained automaton is determinized and the semi linear set is deduced from the length of the obtained lasso if any (notice that since the automaton is deterministic and its alphabet is a singleton, its form will be either a lasso or a simple path.) After this step, there will be no membership predicates involving u.

$$\textsc{Reg-Len} : \frac{Len(\mathcal{R}_1 \cap \ldots \cap \mathcal{R}_m, u) \wedge \phi}{u \in \mathcal{R}_1 \wedge \ldots \wedge u \in \mathcal{R}_m \wedge \phi}\ \Gamma(\phi, u)$$

4.4 Propagating Term Lengths

The rule Term-Leng is the only inference rule in the fourth group. It substitutes the expression $|tr| + |tr'|$ for every occurrence in ϕ of the expression $|tr \cdot tr'|$.

$$\textsc{Term-Leng} : \frac{\phi[|tr \cdot tr'|/|tr| + |tr'|]}{\phi}\ |tr \cdot tr'| \text{ appears in } \phi$$

We can also add rules to systematically add the length predicate $|tr| = |tr'|$ each time an equality $tr = tr'$ appears in a formula; however, such rules are not necessary for the completeness of our procedure, as shown in the next section.

5 Completeness of the Procedure

In this section, we define a class of formulae of *acyclic form* (we say a formula is in acyclic form, or acyclic for short) for which the decision procedure in Section 4 is guaranteed to terminate. For simplicity, we assume w.l.o.g that the formula is a conjunction of predicates and negated predicates.

Non-termination may be caused by an infinite chain of applications of rule Eq-Var of Section 4.2 for removing equalities. Consider for instance the equality $u \cdot v = v \cdot u$. One of the cases generated within the disjunct $\text{Split}'_{\text{Eq-Var-1}}$ of Eq-Var is $v_1 \cdot v_2 = v_2 \cdot v_1$. This is the same as the original equality up to renaming of variables. In this case, the process of removing equalities clearly does not terminate. To prevent this, we will require that no variable can appear on both sides of an equality. We also need to prevent the repetition of a variable inside one side of an equality. This is needed in cases like $u \cdot u = v \cdot v$ where $\text{Split}'_{\text{Eq-Var-1}}$ includes $v_1 = v_2 \cdot v_1 \cdot v_2$, with a variable v_1 on both sides of the equality, which is the situation which we wanted to prevent at the first place. These restrictions must hold initially and must be preserved by applications of any of the rules presented in Sections 4. Attention must be given to rules that modify equalities. Rules such as Eq-Var involve substitution of a variable from one side of an equality by a term from the other side. We need to prevent *chains* of such substitutions that cause variables to appear several times in a (dis)equality. Acyclic formulae must also guarantee that the undesired cases cannot appear after a use of Diseq-Split of Section 4.1 that transforms a disequality to equalities. We respectively state preservation of these restriction and termination of the procedure of Section 4 in theorems 1 and 2 at the end of this Section. First, we need some definitions.

Linear formulae. A formula in $\mathcal{E}_{e,r,l}$ is said to be *linear* if it contains no equality or disequality where a variable appears more than once.

Given a conjunction ϕ in $\mathcal{E}_{e,r,l}$ involving m (dis)equalities, we can build a *dependency graph* $G_\phi = (N, E, \texttt{label}, \texttt{map})$ in the following way. We order the (dis)equalities from e_1 to e_m, where each e_j is of the form $\texttt{lhs}(j) \approx \texttt{rhs}(j)$ for $j : 1 \leq j \leq m$ and $\approx \in \{=, \neq\}$. For each $j : 1 \leq j \leq m$, a node n_{2j-1} is used to refer to the left-hand side of the j^{th} (dis)equality, and n_{2j} to its right-hand side. For example, two different nodes are used even in the case of the simple equality $u = u$, one to refer to the left-hand side, and the other to refer the right-hand side. N is then the set of $2 \times m$ nodes $\{n_i | i : 1 \leq i \leq 2 \times m\}$. The mapping $\texttt{label}$ associates the term $\texttt{lhs}(j)$ (resp. $\texttt{rhs}(j)$) to each node n_{2j-1} (resp. n_{2j}) for $j : 1 \leq j \leq m$. $\texttt{label}$ is not necessarily a one to one mapping. The mapping $\texttt{map} : E \to \{\texttt{rel}, \texttt{var}\}$ labels edges as follows: $\texttt{map}(n, n') = \texttt{rel}$ for each $(n, n') = (n_{2j-1}, n_{2j})$ for each $j : 1 \leq j \leq m$, and $\texttt{map}(n, n') = \texttt{var}$ iff $n \neq n'$, and $\texttt{label}(n)$ and $\texttt{label}(n')$ have some common variables. By construction, $\texttt{map}$ is defined to be total, i.e., E contains only edges that are labeled by $\texttt{map}$.

A *dependency cycle* in $G_\phi = (N, E, \texttt{label}, \texttt{map})$ is a cycle where successive edges have alternating labels. Formally, a dependency cycle is a sequence of distinct nodes $n_0, n_1, \ldots, n_k$ in N with $k \geq 1$ such that 1) for every $i : 0 \leq i \leq k$,

$\texttt{map}(n_i, n_{i+1\%(k+1)})$ is defined, and 2) for each $i : 0 \leq i < k$, $\texttt{map}(n_i, n_{i+1}) \neq \texttt{map}(n_{i+1}, n_{i+2\%(k+1)})$.

Acyclic graph. A conjunction ϕ in $\mathcal{E}_{e,r,l}$ is said to be acyclic iff it is linear and its dependency graph does not contain any dependency cycle.

Theorem 1. *Application of rules of Section 4 preserves acyclicity.*

An *ordered procedure* is any procedure that applies the rules of Section 4 on a formula in $\mathcal{E}_{e,r,l}$ in the four following phases. In the first phase, all disequalities are eliminated using Diseq-Split and Not-Eq. In the second phase, the procedure eliminates one equality at a time by repeatedly applying Eq-Var, Eq-Word and Eq. In the third phase, membership predicates are eliminated by repeatedly applying Reg-Neg, Memb, Not-Memb, Reg-Split and Reg-Len. In the last phase, arithmetic predicates are solved using a standard decision procedure [3].

Theorem 2. *Ordered procedures terminate on acyclic formulae.*

6 Complete Verification of String-Processing Programs

The analysis of string-processing programs has gained importance due to the increased use of string-based APIs and protocols, for instance in the context of databases and Web programming. Much of the existing work has focused on the detection of bugs or the synthesis of attacks; in contrast, the work presented in this paper primarily targets verification of *functional correctness.* The following sections outline how we use our logic $\mathcal{E}_{e,r,l}$ for this purpose. On the one hand, our solver is designed to handle the satisfiability checks needed when constructing finite abstractions of programs, with the help of predicate abstraction [11,13] or Impact-style algorithms [19]; since $\mathcal{E}_{e,r,l}$ can express both length properties and regular expressions, it covers predicates sufficient for a wide range of verification applications. On the other hand, we propose a constraint-based Craig interpolation algorithm for the automatic refinement of program abstractions (Section 6.2), leading to a completeness result in the style of [16]. We represent programs in the framework of Horn clauses [20,12], which make it easy to handle language features like recursion; however, our work is in no way restricted to this setting.

6.1 Horn Constraints with Strings

In our context, a *Horn clause* is a formula $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$ where C is a formula (constraint) in $\mathcal{E}_{e,r,l}$; each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in \mathcal{R}$ to first-order terms; H is either an application $p(t_1, \ldots, t_k)$ of $p \in \mathcal{R}$ to first-order terms, or the constraint *false*. H is called the *head* of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body.* A set $\mathcal{HC}$ of Horn clauses is called *solvable* if there is an assignment that maps every n-ary relation symbol p to a word formula $C_p[x_1, \ldots, x_n]$ with n free variables, such that every clause in $\mathcal{HC}$ is valid. Since Horn clauses can capture properties such as initiation and consecution of invariants, programs can be encoded as sets of Horn clauses in such a way that the clauses are solvable if and only if the program is correct.

Example 1. The example from Section 2 is represented by the following set of Horn clauses, encoding constraints on the intermediate assertions Pre, P_1, P_2, P_3. Note that the clauses closely correspond to the verification conditions given in Fig. 2. Any solution of the Horn clauses represents a set of mutually inductive invariants, and witnesses correctness of the program.

$$\begin{aligned} Pre(s) &\leftarrow true \\ P_1(s') &\leftarrow s' = \epsilon \wedge Pre(s) \\ P_2(s) &\leftarrow P_1(s) \\ P_2(\text{"}a\text{"} \cdot s \cdot \text{"}b\text{"}) &\leftarrow P_2(s) \end{aligned} \qquad \begin{aligned} P_3(s) &\leftarrow P_2(s) \\ false &\leftarrow s \in (a|b)^* \cdot ba \cdot (a|b)^* \wedge P_3(s) \\ false &\leftarrow \forall k.\ 2k \neq |s| \wedge P_3(s) \end{aligned}$$

Algorithms to construct solutions of Horn clauses with the help of *predicate abstraction* have been proposed for instance in [12]; in this context, automatic solving is split into two main steps: 1) the synthesis of *predicates* as building blocks for solutions, and 2) the construction of solutions as Boolean combinations of the predicates. The second step requires a solver to decide consistency of sets of predicates, as well as implication between predicates (a set of predicates implies some other predicate); our logic is designed for this purpose.

$\mathcal{E}_{e,r,l}$ covers a major part of the string operations commonly used in software programs; further operations can be encoded elegantly, including:

- *extraction of substring* v of length *len* from a string u, starting at position *pos*, which is defined by the formula:

$$u = p \cdot v \cdot s \wedge |v| = len \wedge |p| = pos$$

- *replacement* of the substring v (of length *len*, starting at position *pos*) by v', resulting in the new overall string u':

$$u = p \cdot v \cdot s \wedge u' = p \cdot v' \cdot s \wedge |v| = len \wedge |p| = pos$$

- *search* for the first occurrence of a string, using either equations or regular expression constraints.

6.2 Constraint-Based Craig Interpolation

In order to synthesize new predicates for verification, we propose a constraint-based *Craig interpolation* algorithm [6]. We say that a formula $I[\bar{s}]$ is an interpolant of a conjunction $A[\bar{s}], B[\bar{s}]$ over common variables $\bar{s} = s_1, \ldots, s_n$ (and possibly including further local variables), if the conjunctions $A[\bar{s}] \wedge \neg I[\bar{s}]$ and $B[\bar{s}] \wedge I[\bar{s}]$ are unsatisfiable. In other words, an interpolant $I[\bar{s}]$ is an over-approximation of $A[\bar{s}]$ that is disjoint from $B[\bar{s}]$. It is well-known that interpolants are suitable candidates for predicates in software model checking; for a detailed account on the use of interpolants for solving Horn clauses, we refer the reader to [22].

Our interpolation procedure is shown in Alg. 1, and generates interpolants in the form of regular constraints separating $A[\bar{s}]$ and $B[\bar{s}]$. This means that

Algorithm 1. Constraint-based interpolation of string formulae

Input: Interpolation problem $A[\bar{s}] \wedge B[\bar{s}]$ with common variables $\bar{s}$; bound L.
Output: Interpolant $s_1|s_2|\cdots|s_n \in \mathcal{R}$; or result **Inseparable**.

```
1  Aw ← ∅;  Bw ← ∅;
2  while there is RE R of size ≤ L such that Aw ⊆ L(R) and Bw ∩ L(R) = ∅ do
3      if A[s̄] ∧ ¬(s1|s2|···|sn ∈ R) is satisfiable with assignment η then
4          Aw ← Aw ∪ {η(s1)|···|η(sn)};
5      else if B[s̄] ∧ (s1|s2|···|sn ∈ R) is satisfiable with assignment η then
6          Bw ← Bw ∪ {η(s1)|···|η(sn)};
7      else
8          return s1|s2|···|sn ∈ R;
9      end
10 end
11 return Inseparable;
```

interpolants are not arbitrary formulae in the logic $\mathcal{E}_{e,r,l}$, but are restricted to the form $s_1|s_2|\cdots|s_n \in \mathcal{R}$, where "|" $\in \Sigma$ is a distinguished separating letter, and $\mathcal{R}$ is a regular expression. In addition, only interpolants up to a *bound* L are considered; L can limit, for instance, the length of the regular expression $\mathcal{R}$, or the number of states in a finite automaton representing $\mathcal{R}$.

Alg. 1 maintains finite sets Aw and Bw of words representing solutions of $A[\bar{s}]$ and $B[\bar{s}]$, respectively. In line 2, a candidate interpolant of the form $s_1|s_2|\cdots|s_n \in \mathcal{R}$ is constructed, in such a way that $\mathcal{L}(\mathcal{R})$ is a superset of Aw but disjoint from Bw. The concrete construction of candidate interpolants of size $\leq L$ can be implemented in a number of ways, for instance via an encoding as a SAT or SMT problem (as done in our implementation), or with the help of learning algorithms like L^* [1]. It is then checked whether $s_1|s_2|\cdots|s_n \in \mathcal{R}$ satisfies the properties of an interpolant (lines 3, 5), which can be done using the string solver developed in this paper. If any of the properties is violated, the constructed satisfying assignment η gives rise to a further word to be included in Aw or Bw.

Lemma 2 (Correctness). *Suppose bound L is chosen such that it is only satisfied by finitely many formulae $s_1|s_2|\cdots|s_n \in \mathcal{R}$. Then Alg. 1 terminates and either returns a correct interpolant $s_1|s_2|\cdots|s_n \in \mathcal{R}$, or reports **Inseparable**.*

By iteratively increasing bound L, eventually a regular interpolant for any (unsatisfiable) conjunction $A[\bar{s}] \wedge B[\bar{s}]$ can be found, provided that such an interpolant exists at all. This scheme of bounded interpolation is suitable for integration in the complete model checking algorithm given in [16]: since only finitely many predicates can be inferred for every value L, divergence of model checking is impossible for any fixed L. By globally increasing L in an iterative manner, eventually every predicate that can be expressed in the form $s_1|s_2|\cdots|s_n \in \mathcal{R}$ will be found.

7 Implementation

We have implemented our algorithm in a tool called Norn[1] The tool takes as input a formula in the logic described in Section 3, and returns either *Sat* together with a witness of satisfiability (i.e., concrete string values for all variables), or *Unsat*. Norn first converts the given formula to DNF, after which each disjunct goes through the following steps:

1. Recursively split equalities, backtracking if necessary, until no equality constraints are left.
2. Recursively split membership constraints, again backtracking if necessary, and compute the language of each variable. From the language, we extract length constraints which we add to the formula.
3. Solve the remaining length constraints using Princess [3].

We will now explain the second step in more detail. Assume that we have a membership constraint $tr \in A$, where A is an automaton (Norn makes use of dk.brics.automaton [21] for all automata operations). We can remove a sequence of trailing constants $a_1 a_2 \cdots a_k$ in $tr = tr' \cdot a_1 a_2 \cdots a_n$ by replacing the constraint with $tr' \in rev(\delta_{a_k \cdots a_2 a_1}(rev(A)))$, where $\delta_s(A)$ denotes the derivative of A w.r.t. the string s, and $rev(A)$ denotes the reverse of A. We now have a membership constraint $s_1 \cdots s_n \in A'$ where the term consists of a number of segments s_i, each of the form $a_1 \cdots a_{n_i} X_i$, i.e., a number of constants followed by a variable. The procedure keeps, at each step, a mapping m that maps each variable to an automaton representing the language it admits. For the constraint to be satisfiable, the constraints $s_1 \in A'_1$ and $s_2 \cdots s_n \in A'_2$ must be satisfiable for some pair (A_1, A_2) in the splitting of A'. This means that we can update our mapping by $m(X_i) = m(X_i) \cap \delta_{a_1 \cdots a_{n_i}}(A_1)$ and recurse on $s_2 \cdots s_n \in A'_2$. If at any point any automaton in the mapping becomes empty, the membership constraint is unsatisfiable, and we backtrack.

If, in the third step, Princess tells that the given formula is satisfiable, it gives concrete lengths for all variables. By restricting each variable to the solution given by Princess and reversing the substitutions performed in step 1, we can compute witnesses for the variables in the original formula.

Norn can be used both as a library and as a command line tool. In addition to the logic in Section 3, Norn supports character ranges (e.g. $[a - c]$) and the wildcard character (.) in regular expressions. It also supports the divisibility predicate $x \; div \; y$, which says that x divides y. This translates to the arithmetic constraint $x = y * n$, where n is a free variable.

Model Checking. We have integrated Norn into the predicate abstraction-based model checker Eldarica [14], on the basis of the algorithm and interpolation procedure from Section 6. We use the regular interpolation procedure from Section 6.2 in combination with an ordinary interpolation procedure for Presburger arithmetic to infer predicates about word length. Table 1 gives an overview

[1] Available at `http://user.it.uu.se/~jarst116/norn/`.

Table 1. Verification runtime for a set of string-processing programs. Experiments were done on an Intel Core i5 machine with 3.2GHz, running 64 bit Linux.

Program	Property	Time
$a^n b^n$ (Fig. 1)	$s \notin (a+b)^* \cdot ba \cdot (a+b)^* \wedge \exists k.\ 2k = \lvert s \rvert$	8.0s
StringReplace	pre: $s \in (a+b+c)^*$; post: $s \in (a+c)^*$	4.5s
ChunkSplit	pre: $s \in (a+b)^*$; post: $s \in (a+b+c)^*$	5.5s
Levenshtein	$dist \leq \lvert s \rvert + \lvert t \rvert$	5.3s
HammingDistance	$dist = \lvert v \rvert$ if $u \in 0^*, v \in 1^*$	27.1s

of preliminary results obtained when analyzing a set of hand-written string-processing programs. Although the programs are quite small, the presence of string operations makes them intricate to analyze using automated model checking techniques; most of the programs require invariants in form of regular expressions for verification to succeed. Our implementation is able to verify all programs fully automatically within a few seconds; since performance has not been the main focus of our implementation work so far, further optimization will likely result in much reduced runtimes. To the best of our knowledge, all of the programs are beyond the scope of other state-of-the-art software model checkers.

8 Conclusions and Future Work

In contrast to much of the existing work that has focused on the detection of bugs or the synthesis of attacks for string-manipulating programs; the work presented in this paper primarily targets verification of *functional correctness.* To achieve this goal, we have made several key contributions. First, we have presented a decision procedure for a rich logic of strings. Although the problem in its generality remains open, we are able to identify an expressive fragment for which our procedure is both sound and complete. We are not aware of any decision procedure with a similar expressive power. Second, we leverage on the fact that our logic is able to reason both about length properties and regular expressions in order to capture and manipulate predicates sufficient for a wide range of verification applications. Future works include experimenting with better integrations of the different theories, exploring different Craig interpolation techniques, and exploring the applicability of our framework to more general classes of string processing applications.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
2. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with slam. Commun. ACM 54(7), 68–76 (2011)
3. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. Journal of Automated Reasoning 47, 341–367 (2011)

4. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. Z. Math. Logik Grundlagen Math. 34(4) (1988)
5. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
6. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. The Journal of Symbolic Logic 22(3) (1957)
7. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on CAD of Integrated Circuits and Systems 27(7), 1165–1178 (2008)
9. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
10. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: What's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) HVC. LNCS, vol. 7857, pp. 209–226. Springer, Heidelberg (2013)
11. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
12. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416 (2012)
13. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: 31st POPL (2004)
14. Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 247–251. Springer, Heidelberg (2012)
15. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. 41(4) (2009)
16. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
17. Kieżun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Transactions on Software Engineering and Methodology 21(4) (2012)
18. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Mathematics of the USSR-Sbornik 32(2), 129–198 (1977)
19. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
20. Méndez-Lojo, M., Navas, J., Hermenegildo, M.V.: A flexible (C)LP-based approach to the analysis of object-oriented programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 154–168. Springer, Heidelberg (2008)
21. Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2010), `http://www.brics.dk/automaton/`
22. Rümmer, P., Hojjat, H., Kuncak, V.: Classifying and solving horn clauses for verification. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 1–21. Springer, Heidelberg (2014)

23. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A Symbolic Execution Framework for JavaScript. In: IEEE Symposium on Security and Privacy, pp. 513–528. IEEE Computer Society (2010)
24. Saxena, P., Hanna, S., Poosankam, P., Song, D.: FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In: NDSS. The Internet Society (2010)
25. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010)
26. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A Z3-based string solver for web application analysis. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 114–124. ACM, New York (2013)

All for the Price of Few
(Parameterized Verification through View Abstraction)

Parosh Aziz Abdulla[1], Frédéric Haziza[1], and Lukáš Holík[1,2]

[1] Uppsala University, Sweden
[2] Brno University of Technology, Czech Republic

Abstract. We present a simple and efficient framework for automatic verification of systems with a parameteric number of communicating processes. The processes may be organized in various topologies such as words, multisets, rings, or trees. Our method needs to inspect only a small number of processes in order to show correctness of the whole system. It relies on an abstraction function that views the system from the perspective of a fixed number of processes. The abstraction is used during the verification procedure in order to dynamically detect cut-off points beyond which the search of the state space need not continue. We show that the method is complete for a large class of well quasi-ordered systems including Petri nets. Our experimentation on a variety of benchmarks demonstrate that the method is highly efficient and that it works well even for classes of systems with undecidable verification problems.

1 Introduction

We address verification of safety properties for *parameterized systems* that consist of arbitrary numbers of components (processes) organized according to a regular pattern. The task is to perform *parameterized verification*, i.e., to verify correctness regardless of the number of processes. This amounts to the verification of an infinite family; namely one for each possible size of the system. The term *parameterized* refers to the fact that the size of the system is (implicitly) a parameter of the verification problem. Parameterized systems arise naturally in the modeling of mutual exclusion algorithms, bus protocols, distributed algorithms, telecommunication protocols, and cache coherence protocols. For instance, the specification of a mutual exclusion protocol may be parameterized by the number of processes that participate in a given session of the protocol. In such a case, it is interesting to verify correctness regardless of the number of participants in a particular session. As usual, the verification of safety properties can be reduced to the problem of checking the reachability of a set of *bad configurations* (the set of configurations that violate the safety property).

Existing approaches. An important approach to parameterized verification has been *regular model checking* [25,5,9] in which regular languages are used as symbolic representations of infinite sets of system configurations, and automata-based techniques are employed to implement the verification procedure. The main problem with such techniques is that they are heavy since they usually rely on several layers of computationally expensive automata-theoretic constructions, in many cases leading to a state space

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 476–495, 2013.

explosion that severely limits their applicability. Another class of methods analyze *approximated* system behavior through the use of abstraction techniques. Such methods include *counter abstraction* [22,30], *invisible invariant* generation [6,31], *environment abstraction* [11], and *monotonic abstraction* [3] (see Section 7).

In a similar manner to [24], this work is inspired by a strong empirical evidence that parameterized systems often enjoy a *small model property*. More precisely, analyzing only a small number of processes (rather than the whole family) is sufficient to capture the reachability of bad configurations. On the one hand, bad configurations can often be characterized by minimal conditions that are possible to specify through a fixed number of *witness* processes. For instance, in a mutual exclusion protocol, a bad configuration contains *two* processes in their critical sections; and in a cache coherence protocol, a bad configuration contains *two* cache lines in their *exclusive* states. In both cases, having the two witnesses is sufficient to make the configuration bad (regardless of the actual size of the configuration). On the other hand, it is usually the case that such bad patterns (if existing) appear already in small instances of the system, as observed in our experimental section.

Our approach. We introduce a method that exploits the small model property, and performs parameterized verification by only inspecting a small set of *fixed* instances of the system. Furthermore, the instances that need to be considered are often small in size (typically three or four processes) which allows for a very efficient verification procedure. The framework can be applied uniformly to generate *fully automatic* verification algorithms for wide classes of parameterized systems including ones that operate on linear, ring, or tree-like topologies, or systems that contain unbounded collections of anonymous processes (the latter class is henceforth referred to as having a *multi-set* topology).

At the heart of the method is an operation that allows to detect *cut-off* points beyond which the verification procedure need not continue. Intuitively, reaching a cut-off point means that we need not inspect larger instances of the system: the information collected so far during the exploration of the state space allows us to conclude safely that no bad configurations will occur in the larger instances. The cut-off analysis is executed *dynamically* in the sense that it is performed on-the-fly during the verification procedure itself. It is based on an abstraction function, called *view abstraction*, parameterized by a constant k, and it approximates a configuration by the set of all its projections containing at most k processes. We call the sub-configurations *views*. For instance, when a configuration is a word of process states (represented as an array of processes), its abstraction is the set of all its subwords of length at most k. Furthermore, for a given set of views X, its concretization, denoted as $\gamma_k(X)$, is the set of configurations (of any size) for which *all* their views belong to X.

The verification method performs two search procedures in parallel. The first performs a standard (explicit-state) forward reachability analysis trying to find a bad configuration among system configurations of size k (for some natural number k). If a bad configuration is encountered then the system is not safe. The second procedure performs a *symbolic* forward reachability analysis in the abstract domain of sets of views of size at most k. When the computation terminates, it will have collected an over-approximation of all views of size up to k of all reachable configurations (of all sizes).

If there is no bad configuration in the concretization of this set, then a cut-off point has been found and the system can be claimed safe. If neither of the parallel procedures reaches a conclusion during iteration k, the value of k is increased by one (thus increasing the precision of the abstraction). Notice that the abstract search requires computing the *abstract post-image* of a set X of views of size at most k, which is the set X' of views (of size at most k) of successors of $\gamma_k(X)$. Obviously, this cannot be performed straightforwardly since the set of configurations $\gamma_k(X)$ is infinite. A crucial contribution of the paper is to show that, for all the classes of parameterized systems that we consider, it is sufficient to only compute successors of configurations from $\gamma_k(X)$ that are of the size at most $k+\ell$, where ℓ is a small constant, typically 1. Intuitively, the reason is that the precondition for firing a transition is the presence of a *bounded* number of processes in certain states. The views need only to encompass these processes in order to determine the successor view. This property is satisfied by a wide class of concurrent systems including the ones we consider in this paper. For instance, in rendez-vous communication between a pair of processes, the transition is conditioned by the states of *two* processes; in broadcast communication, *one* process initiates the transition (while the other processes may be in any state); in existential global transitions (see below), we need *two* processes, namely the witness and the process performing the transition; in Petri nets, the number of required processes is bounded by the in-degree of the transitions (which is fixed for a given Petri net), etc. We will show formally that this property is satisfied by all the types of transitions we consider.

Applications. We have instantiated the method to obtain automatic verification procedures for four classes of parameterized systems, namely systems where the processes are organized as arrays, rings, trees, or multisets. Each instantiation is straightforward and is achieved by defining the manner in which we define the views of a configuration. More precisely, these views are (naturally) defined as subwords, cyclic subwords, subtrees, resp. subsets for the above four classes. Once the views are fixed we obtain a fully automatic procedure for all parameterized systems in the class. In the systems we consider, we allow a rich set of features, in which processes may perform local transitions, rendez-vous, broadcasts, and universally or existentially guarded transitions. In a universally guarded transition, the process checks whether the states of *all* other processes inside the system satisfy a given constraint before it performs the transition. In an existentially quantified transition, the processes checks that there is *at least one* other process satisfying the condition. Furthermore, we allow dynamic behaviors such as the creation and deletion of processes during the execution of the system.

In the basic variant of our method, we assume that existential and universal global conditions of transitions are checked atomically. The same assumption is made in many landmark works on parameterized systems (e.g. [11,31,10,5,6,29,3]). However, actual implementations of global checks are usually not atomic. They are typically implemented as for-loops ranging over indices of processes. Iterations of such a loop may be interleaved with transitions of other processes, therefore modeling the loop as an atomic transition means under-approximating the behavior of the system. Verification of systems with non-atomic global checks is significantly harder. It requires to distinguish intermediate states of a for-loop performed by a process. Their number is proportional to the number of processes in the system. Moreover, any number of processes may be

performing a for-loop at the same time. As we will show, our method can be easily adapted to this setting, while retaining its simplicity and efficiency.

Implementation. We have implemented a prototype based on the method and run it on a wide class of benchmarks, including mutual exclusion protocols on arrays (e.g., Burns', Szymanski's, and Dijkstra's protocols), cache coherent protocols (e.g., MOSI and German's protocol), different protocols on tree-like architectures (e.g. percolate, arbiter, and leader election), ring protocols (token passing), and different Petri nets.

The class of systems we consider have undecidable reachability properties, and hence our method is necessarily incomplete (the verification procedure is not guaranteed to terminate in case the safety property is satisfied). However, as shown by our experimentation, the tool terminates efficiently on all the tested benchmarks.

Completeness. Although the method is not complete in general, we show that is complete for a large class of systems, namely those that induce *well quasi-ordered* transition systems [2,1] and satisfy certain additional technical requirements. This implies that our method is complete for e.g., Petri nets. Notice that, as evident from our experiments, the method can in practice handle even systems that are outside the class.

Outline. To simplify the presentation, we instantiate our framework in a step-wise manner. In Section 2, we introduce our model for parameterized systems operating on linear topologies and describe our verification method in Section 3. In Section 4, we describe how the framework can be extended to incorporate other kinds of transitions such as broadcast, rendez-vous, dynamic process deletion/creation, and non-atomic checks of global conditions; and to cover other classes of topologies such as ring, multiset, and tree-like structures. The completeness of our method for well quasi-ordered systems is shown in Section 5. We report on our experimental results in Section 6, and describe related work in Section 7. Finally, we give some conclusions and directions for future research in Section 8.

2 Parameterized Systems

We introduce a standard notion of a parameterized system operating on a linear topology, where processes may perform local transitions or universally/existentially guarded transitions (this is the standard model used e.g. in [31,11,3,29]).

A parameterized system is a pair $\mathcal{P} = (Q, \Delta)$ where Q is a finite set of *local states* of a process and Δ is a set of *transition rules* over Q. A transition rule is either *local* or *global*. A local rule is of the form $s \to s'$, where the process changes its local state from s to s' independently of the local states of the other processes. A global rule is of the form **if** $\mathbb{Q} j \circ i : S$ **then** $s \to s'$, where $\mathbb{Q} \in \{\exists, \forall\}$, $\circ \in \{<, >, \neq\}$ and $S \subseteq Q$. Here, the ith process checks also the local states of the other processes when it makes the move. For instance, the condition $\forall j < i : S$ means that "for every j such that $j < i$, the jth process should be in a local state that belongs to the set S"; the condition $\forall j \neq i : S$ means that "all processes except the ith one should be in local states that belong to the set S"; etc.

A parameterized system $\mathcal{P} = (Q, \Delta)$ induces a *transition system* (TS) $\mathcal{T} = (C, \to)$ where $C = Q^*$ is the set of its *configurations* and $\to \subseteq C \times C$ is the *transition relation*.

We use $c[i]$ to denote the state of the ith process within the configuration c. The transition relation $\rightarrow$ contains a transition $c \rightarrow c'$ with $c[i] = s$, $c'[i] = s'$, $c[j] = c'[j]$ for all $j : j \neq i$ iff either (i) Δ contains a local rule $s \rightarrow s'$, or (ii) Δ contains a global rule **if** $\mathbb{Q} j \circ i : S$ **then** $s \rightarrow s'$, and one of the following conditions is satisfied:

- $\mathbb{Q} = \forall$ and for all $j : 1 \leq j \leq |c|$ such that $j \circ i$, it holds that $c[j] \in S$.
- $\mathbb{Q} = \exists$ and there exists $j : 1 \leq j \leq |c|$ such that $j \circ i$ and $c[j] \in S$.

An instance of the *reachability problem* is defined by a parameterized system $\mathcal{P} = (Q, \Delta)$, a regular set $I \subseteq Q^+$ of *initial configurations*, and a set $Bad \subseteq Q^+$ of *bad configurations*. Let $\sqsubseteq$ be the usual *subword relation*, i.e., $u \sqsubseteq s_1 \ldots s_n$ iff $u = s_{i_1} \ldots s_{i_k}, 1 \leq i_1 \ldots i_k \leq n$ and $i_j < i_{j+1}$ for all $j : 1 \leq j < k$. We assume that Bad is the upward closure $\{c \mid \exists b \in B : b \sqsubseteq c\}$ of a given *finite* set $B \subseteq Q^+$ of *minimal bad configurations*. This is a common way of specifying bad configurations which often appears in practice, see e.g. the running example of Burn's mutual exclusion protocol below. We say that $c \in C$ is *reachable* iff there are $c_0, \ldots, c_l \in C$ such that $c_0 \in I$, $c_l = c$, and $c_i \rightarrow c_{i+1}$ for all $0 \leq i < l$. We use $\mathcal{R}$ to denote the set of all reachable configurations. We say that the system $\mathcal{P}$ is *safe* w.r.t. I and Bad if no bad configuration is reachable, i.e. $\mathcal{R} \cap Bad = \emptyset$.

We define the *post-image* of a set $X \subseteq C$ to be the set $post(X) := \{c' \mid c \rightarrow c' \wedge c \in X\}$. For $n \in \mathbb{N}$ and a set of configurations $S \subseteq C$, we use S_n to denote its subset $\{c \in S \mid |c| \leq n\}$ of configurations of size up to n.

Running example. We illustrate the notion of a parameterized systems with the example of Burns' mutual exclusion protocol [26]. The protocol ensures exclusive access to a shared resource in a system consisting of an unbounded number of processes organized in an array. The pseudocode of the process at the ith position of the array and the transition rules of the parameterized system are given in Figure 1. A state of the ith process consists of a program location and a value of the local variable $flag[i]$. Since the value of $flag[i]$ is invariant at each location, states correspond to locations.

A configuration of the induced transition system is a word over the alphabet $\{1, \ldots, 6\}$ of local process states. The task is to check that the protocol guarantees exclusive access to the shared resource (line 6) regardless of the number of processes. A configuration is considered to be bad if it contains two occurrences of state 6, i.e., the set of minimal bad configurations B is $\{6\,6\}$. Initially, all processes are in state 1, i.e. $I = 1^+$.

Burns(i)

1 $flag[i] := 0$;
2 **if** $\exists j < i : flag[i] = 1$ **then goto** 1;
3 $flag[i] := 1$;
4 **if** $\exists j < i : flag[i] = 1$ **then goto** 1;
5 **await** $\forall j > i : flag[j] \neq 1$;
6 $flag[i] := 0$; **goto** 1 CS

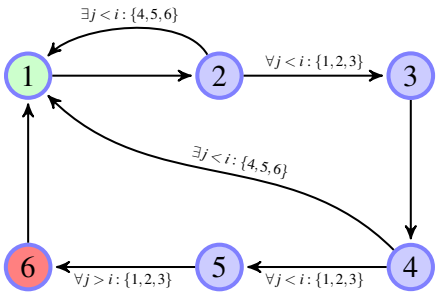


Fig. 1. Pseudocode and transition rules of Burns' protocol

3 Verification Method

In this section, we describe our verification method instantiated to the case of parameterized systems described in Section 2. First, we describe the abstraction we use, then we present the procedure.

3.1 View Abstraction

We abstract a configuration c by a set of *views* each of which is a subword of c. The *abstraction function* $\alpha_k : C \to 2^{C_k}$ maps a configuration c into the set $\alpha_k(c) = \{v \in C_k \mid v \sqsubseteq c\}$ of all its views (subwords) of size up to k. We lift α_k to sets of configurations as usual. For every $k \in \mathbb{N}$, the *concretization* function $\gamma_k : 2^{C_k} \to 2^C$ inputs a set of views $V \subseteq C_k$, and returns the set of configurations that can be reconstructed from the views in V. In other words, $\gamma_k(V) = \{c \in C \mid \alpha_k(c) \subseteq V\}$.

Abstract post-image. As usual, the *abstract post-image* of a set of views $V \subseteq C_k$ is defined as $Apost_k(V) = \alpha_k(post(\gamma_k(V)))$. Computing $Apost_k(V)$ is a central component of our verification procedure. It cannot be computed straightforwardly since the set $\gamma_k(V)$ is typically infinite. As a main contribution of the paper, we show that it is sufficient to consider only those configurations in $\gamma_k(V)$ whose sizes are up to $k+1$. There are finitely many such configurations, and hence their post-image can be computed. Formally, for $\ell \geq 0$, we define $\gamma_k^\ell(V) := \gamma_k(V) \cap C_\ell$ and show the following *small model* lemma for the class of systems of Section 2. We will show similar lemmas for the other classes of systems that we present in the later sections.

Lemma 1. *For any $k \in \mathbb{N}$ and $X \subseteq C_k$, $\alpha_k(post(\gamma_k(X))) \cup X = \alpha_k(post(\gamma_k^{k+1}(X))) \cup X$.*

The property of the transition relation which allows us to prove the lemma is that, loosely speaking, the transitions have *small preconditions*. That is, there is a transition that can be fired from a configuration c and generate a view $v \in C_k$ only if c contains a certain view v' of some limited size, here up to $k+1$.

Running Example. Consider for instance the set $V = \{1,2,3,4,6,12,16,32,34,42\} \subseteq C_2$ of views of Burns' protocol. We will illustrate that we need to reason only about configurations of $\gamma_2(V)$, which are of size at most 3, to decide which views belong to $Apost_2(V)$.

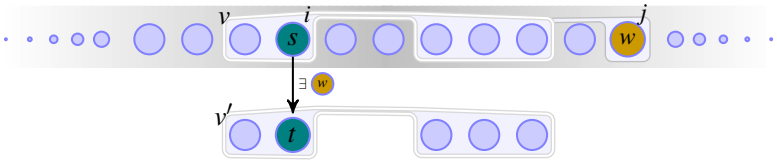
Take the existentially guarded transition $2 \to 1$. It can be fired only from configurations that contain 2 together with a witness from $\{4,5,6\}$ on the left. $Apost_2(V)$ contains the view 31 since $\gamma_2^{2+1}(V)$ contains 342 from where the existential transition $2 \to 1$ can be fired. (342 belongs to $\gamma_2(V)$ because all its views 2, 3, 4, 32, 34, and 42 are in V). It does not contain the view 22 since 12 cannot be completed by the needed witness (12 cannot be extended by, e.g., 6 since V does not contain 26 and 62).

Consider now the universally guarded transition $2 \to 3$. The transition can be fired only from configurations that contain 2. Since $2 \to 3$ can be fired on $32 \in \gamma_2(V)$, $Apost_2(V)$ contains 33. But it does not contain the view 43 since the universal guard prevents firing $2 \to 3$ on configurations containing 42.

Proof. We present the part of the proof of Lemma 1 which deals with existentially guarded transitions. The parts dealing with local and universally guarded transitions are simpler and are moved to the appendix. We will show that for any configuration $c \in \gamma_k(V)$ of size $m > k+1$ such that there is a transition $c \to c'$ induced by an existentially guarded rule $r \in \Delta$ with $v' \in \alpha_k(c')$, the following holds: Either $v' \in V$ or there is a configuration $d \in \gamma_k(V)$ of size at most $k+1$ with a transition $d \to d'$ induced by r with $v' \in \alpha_k(d')$.

A subset of positions $p = \{i_1, \ldots, i_l\} \subseteq \{1, \ldots, n\}, l \leq k$, with $i_1 < \ldots < i_l$ of a configuration $c = s_1 \ldots s_n$ defines the view $view(c,p) = s_{i_1} \ldots s_{i_l}$ of c. By definition, v' equals $view(c',p)$ for some $p \subseteq \{1, \ldots, m\}$. Let v be $view(c,p)$. Notice that since $c \in \gamma_k(V)$, any view of c of size at least k belongs to $\gamma_k(V)$. Therefore also $v \in \gamma_k(V)$. Let $1 \leq i \leq m$ be the index of the position in which c' differs from c. If $i \notin p$, then $v = view(c,p) = view(c',p) = v'$. In this case, we trivially have $v' \in V$. We can take $d = v$ and $d' = v'$.

Assume now that $i \in p$. Let r be the rule **if** $\exists j \circ i : S$ **then** $s \to t$ where $\circ \in \{<, >, \neq\}$. There are two cases: 1) there is a witness w from S at some position $j \in p$ enabling the transition $c \to c'$. Then v still contains the witness on an appropriate position needed to fire r. Therefore $v \to v'$ is a transition of the system induced by r, and we can take $d = v$ and $d' = v'$. 2) no witness enabling the transition $c \to c'$ is at a position $j \in p$. Then there is no guarantee that $v \to v'$ is a transition of the system. However, the witness enabling the transition $c \to c'$ is at some position $j \in \{1, \ldots, m\}$. We will create a configuration of size at most $k+1$ by including this position j to v, as illustrated in the figure. Let $p' = p \cup \{j\}$. Then $view(c,p') \to view(c',p')$ is a transition of the system induced by r since $view(c,p')$ contains both s and a witness from S at an appropriate position. We clearly have that $v' \in \alpha_k(view(c',p'))$. We also have that $view(c,p') \in \gamma_k(V)$ since $view(c,p') \sqsubseteq c$ and $c \in \gamma_k(V)$. We may therefore take $d = view(c,p')$ and $d' = view(c',p')$. □



3.2 Procedure

Our verification procedure for solving an instance of the verification problem defined in Section 2 is described in Algorithm 1. It performs two search procedures in parallel. Specifically, it searches for a bad configuration reachable from initial configurations of size k; and it searches for a cut-off point k where it derives a set of views $V \subseteq C_k$ such that

(i) V is an invariant for the instances of the system (that is, $\mathcal{R} \subseteq \gamma_k(V)$ and $Apost_k(V) \subseteq V$), and
(ii) which is sufficient to prove $\mathcal{R}$ safe (that is, $\gamma_k(V) \cap Bad = \emptyset$).

Algorithm 1. Verification Procedure

1 **for** $k := 1$ **to** ∞ **do**
2 **if** $\mathcal{R}_k \cap Bad \neq \emptyset$ **then return** Unsafe
3 $V := \mu X.\alpha_k(I) \cup Apost_k(X)$
4 **if** $\gamma_k(V) \cap Bad = \emptyset$ **then return** Safe

For a given k, an invariant V is computed on line 3. Notice that V is well-defined since $\gamma_k, post, \alpha_k$ and hence also $Apost_k$ are monotonic functions for all $k \in \mathbb{N}$ (w.r.t. $\subseteq$). Lemma 2 guarantees that V is indeed an invariant:

Lemma 2. *For any $i \in \mathbb{N}$ and $X \subseteq C_i$, $\alpha_i(I) \subseteq X \wedge Apost_i(X) \subseteq X \implies \alpha_i(\mathcal{R}) \subseteq X$.*

If the system is unsafe, the search on line 2 will eventually discovers a bad configuration. The cut-off condition is tested on line 4. If the test does not pass, then we do not know whether the system is indeed unsafe or whether the analysis has hit a spurious counterexample (due to a too liberal abstraction). Therefore, the algorithm increases precision of the abstraction by increasing k and reiterating the loop. An effective implementation of the procedure requires carrying out the following steps:

1. *Computing the abstraction $\alpha_k(I)$ of initial configurations.* This step is usually easy. For instance, in the case of Burns' protocol, all processes are initially in state 1, hence $\alpha_k(I)$ contains only the words $1^l, l \leq k$. Generally, I is a (very simple) regular set, and $\alpha_k(I)$ is computed using a straightforward automata construction.
2. *Computing the abstract post-image.* Thanks to Lemma 1, the abstract post-image can be computed by applying γ_k^{k+1} (which yields a finite set), *post*, and α_k (in that order).
3. *Evaluating the test $\gamma_k(V) \cap Bad = \emptyset$.* Since *Bad* is the upward closure of a finite set B, the test can be carried out by testing whether there is $b \in B$ such that $\alpha_k(b) \subseteq V$.
4. *Exact reachability analysis.* Line 2 requires the computation of $\mathcal{R}_k$. Since $\mathcal{R}_k$ is finite, this can be done using any procedure for exact state space exploration.

Since the problem is generally undecidable, existence of k for which the test on line 4 succeeds for a safe system cannot be guaranteed and the algorithm may not terminate. However, as discussed in Section 5, such a guarantee can be given under the additional requirement of monotonicity of transition relation w.r.t. a well-quasi ordering. The method terminates otherwise for all our examples discussed in Section 6, many of which are *not* well quasi-ordered.

Running example. When run on Burns' protocol, Algorithm 1 starts by computing $\mathcal{R}_1 = \{1,\ldots,6\}$. Because $\mathcal{R}_1$ does not contain any bad configurations, the algorithm moves onto computing the fixpoint V_1 of line 3. The iteration starts with $X = \alpha_1(I) = \{1\}$ and continues until $X = V_1 = \{1,\ldots,6\}$. The test on line 4 subsequently fails since $\gamma_1(V_1)$ contains 66. Since both tests fail, the first iteration does not allow us to conclude whether the protocol is safe or not, so the algorithm increases the precision of the abstraction by increasing k.

In the second iteration with $k = 2$, $\mathcal{R}_2$ is still safe. The fixpoint computation starts with $X = \alpha_2(I) = \{1, 11\}$. When $Apost_2$ is applied on $\{1, 11\}$, we first construct the set $\gamma_2^{2+1}(\{1, 11\})$ which contains the extension 111 of 11, 11 and 1. Their successors are $2, 12, 21$, and $112, 121, 211$, which are abstracted into $\{1, 2, 11, 12, 21\}$. The fixpoint computation continues with $X = \{1, 2, 11, 12, 21\}$ and constructs the concretization $\gamma_2^3(X) = X \cup \{112, 121, 211\}$. Their successors are $2, 3, 12, 21, 22, 31, 13$, and 122, 212, 221, 113, 131, 311 which are abstracted into the views $1, 2, 3, 11, 12, 21, 22, 31, 13$. The next iteration will start with $X = \{1, 2, 3, 11, 12, 21, 22, 13, 31\}$. The computation reaches, after 8 further iterations, the fixpoint $X = V_2$ which contains all words from $\{1, \ldots, 6\} \cup \{1, \ldots, 6\}^2$ except 65 and 66. This set satisfies the assumptions of Lemma 2, and hence it is guaranteed to contain all views (of size at most 2) of all reachable configurations of the system. Since the view 66 is not present (recall $\alpha_2(Bad) = \{6, 66\}$), no reachable configuration of the system is bad. The algorithm reached the cut-off point $k = 2$ of Burns' protocol, and the system is proved safe.

4 Extensions

In this section, we describe how to extend the class of parameterized systems that we presented in Section 2. The extensions are obtained 1) by extending the types of transition rules that we allow, 2) by replacing transitions with atomically checked global conditions by more realistic for-loops, and 3) by considering topologies other than the linear ones. As we shall see below, the extensions can be handled by our method with straightforward extensions of the method of Section 3.

4.1 More Communication Mechanisms

Broadcast. In a broadcast transition, an arbitrary number of processes change states simultaneously. A broadcast rule is a pair $(s \to s', \{r_1 \to r_1', \ldots, r_m \to r_m'\})$. It is deterministic in the sense that $r_i \neq r_j$ for $i \neq j$. The broadcast is initiated by a process, called the *initiator*, which triggers the transition rule $s \to s'$. Together with the initiator, an arbitrary number of processes, called the *receptors*, change state simultaneously. More precisely, if the local state of a process is r_i, then the process changes its local state to r_i'. Processes whose local states are different from $s, r_1, \ldots r_m$ remain passive during the broadcast. Formally, the broadcast rule induces transitions $c \to c'$ of $\mathcal{T}$ where for some $i : 1 \leq i \leq |c|$, $c[i] = s$, $c'[i] = s'$, and for each $j : 1 \leq j \neq i \leq |c|$, if $c[j] = r_k$ (for some k) then $c'[j] = r_k'$, otherwise $c[j] = c'[j]$.

In a similar manner to globally guarded transitions, broadcast transitions have small preconditions. Namely, to fire a transition, it is enough that an initiator is present in the transition. More precisely, for parameterized systems with local, global, and broadcast transitions, Lemma 1 still holds (in the proof of Lemma 1, the initiator is treated analogously to a witness of an existential transition). Therefore, the verification method from Section 3 can be used without any change.

Rendez-vous. In rendez-vous, multiple processes change their states simultaneously. A *simple rendez-vous* transition rule is a tuple of local rules $\delta = (r_1 \to r_1', \ldots, r_m \to$

$r'_m), m > 1$. Multiple occurrences of local rules with the same source state r in the tuple does not mean non-determinism, but that the rendez-vous requires multiple occurrences of r in the configuration to be triggered. Formally, the rule induces transitions $c \to c'$ of $\mathcal{T}$ such that there are $i_1, \ldots, i_m$ with $i_j \neq i_k$ for all $j \neq k$, such that $c[i_1] \cdots c[i_m] = r_1 \cdots r_m$, $c'[i_1] \cdots c'[i_m] = r'_1 \cdots r'_m$, and $c'[\ell] = c[\ell]$ if $\ell \notin \{i_1, \ldots, i_m\}$.

Additionally, we define a *generalized rendez-vous (or just rendez-vous) transition rules* in order to model *creation and deletion* of processes and also Petri net transitions that change the number of tokens in the net. A generalized rendez-vous rule δ is as a simple rendez-vous rule, but it can in addition to the local rules contain two types of special rules: of the form $\bullet \to r, \bullet \notin Q$ (acting as a placeholder), which are used to model creation of processes, and of the form $r \to \bullet$, which are used to model deletion of processes. When a generalized rendez-vous rule is fired, the starting configuration is first enriched with $\bullet$ symbols in order to facilitate creation of processes by the rule $\bullet \to r$, then the rule is applied as if it was a simple rendez-vous rule, treating $\bullet$ as a normal state (states of the processes that are to be deleted are rewritten to $\bullet$ by the rules $r \to \bullet$). Finally, all occurrences of $\bullet$ are removed. Formally, a generalized rendez-vous rule induces a transition $c \to c'$ if there is $c_\bullet \in \{\bullet\}^* c[1]\{\bullet\}^* \cdots \{\bullet\}^* c[|c|]\{\bullet\}^*$ such that $c_\bullet \to c'_\bullet$ is a transition of the system with states $Q \cup \{\bullet\}$ induced by δ (treated as a simple rendez-vous rule), and c' arises from $c'_\bullet$ by erasing all occurrences of $\bullet$.

Rendez-vous transitions have small preconditions, but unlike existentially quantified transitions, firing a transition may require presence of more than two (but still a fixed number) processes in certain states (the number is the arity of the transition). It essentially corresponds to requiring the presence of more than one witness. This is why Lemma 1 holds here only in the weaker variant:

Lemma 3. *Let Δ contain rules of any previously described type (i.e., local, global, broadcast, rendez-vous), and let $m+1$ is the largest arity of a rendez-vous rule in Δ. Then, for any k and $V \subseteq C_k$, $\alpha_k(post(\gamma_k(V))) \cup V = \alpha_k(post(\gamma_k^{k+m}(V))) \cup V$.*

Global variables. Communication via shared variables is modeled using a special process, called *controller*. Its local state records the state of all shared variables in the system. A configuration of a system with global variables is then a word $s_1 \ldots s_n c$ where $s_1, \ldots, s_n$ are the states of individual processes and c is the state of the controller. An individual process can read and update a shared variable. A read is modeled by a rendez-vous rule of the form $(s \to s', c \to c)$ where c is a state of the controller and s, s' are states of the process. An update is modeled using a rendez-vous rule $(s \to s', c \to c')$.

To verify systems with shared variables of finite domains, we use a variant of the abstraction function which always keeps the state of the controller in the view. Formally, for a configuration wc where $w \subseteq Q^+$ and c is the state of the controller, α_k returns the set of words vc where v is a subword of w of length at most k. The concretization and abstract-post image are then defined analogously as before, based on α_k, Lemma 1 and Lemma 2 still hold. The method of Section 3 can be thus used in the same way as before.

Another type of global variable is a *process pointer*, i.e., a variable ranging over process indices. This is used, e.g., in Dijkstra's mutual exclusion protocol. A process pointer is modeled by a local Boolean flag p for each process state. The value of p is

true iff the pointer points to the process (it is true for precisely one process in every configuration). An update of the pointer is modeled by a rendez-vous transition rule which sets to *false* the flag of the process currently pointed to by the pointer and sets to *true* the flag of the process which is to become the target of the pointer.

4.2 Transitions That Do not Preserve Size

We now discuss the case when the transition relation does not preserve size of configurations, which happens in the case of generalised rendez-vous. $\mathcal{R}_k$ then cannot be computed straightforwardly since computations reaching configurations of the size up to k may traverse configurations of larger sizes. Therefore, similarly as in [21], we only consider runs of the system visiting configurations of the size up to k. That is, on line 2 of Algorithm 1, instead of computing $\mathcal{R}_k = \mu X.\, I_k \cup post(X)$, we compute its under-approximation $\mu X.\, (I \cup post(X)) \cap C_k$. The computation terminates provided that C_k is finite. The algorithm is still guaranteed to return Unsafe if a configuration in *Bad* is reachable, since then there is $k \in \mathbb{N}$ such that the bad configuration is reachable by a finite path traversing configurations of the size at most k.

4.3 Non-atomic Global Conditions

We extend our method to handle systems where global conditions are not checked atomically. We replace both existentially and universally guarded transition rules by a simple variant of a for-loop rule:

$$\textbf{if foreach } j \circ i : S \textbf{ then } q \to r \textbf{ else } q \to s$$

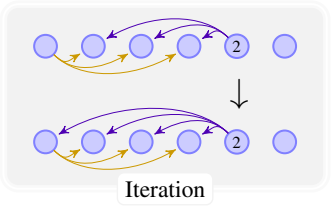
where $q, r, s \in Q$ is resp. a source state, a target state, and an escape state, $\circ \in \{<, >, \neq\}$, and $S \subseteq Q$ is a *condition*. For instance, line 2 of Burns' protocol would be replaced by **if foreach** $j < i : \{1,2,3\}$ **then** $2 \to 3$ **else** $2 \to 1$.

The semantics of a system with for-loop rules is defined as an extension of the transition system from Section 2. Configurations are extended with a binary relation over their positions, that is, a configuration is now a pair $(c, \checkmark)$ where c is a word over Q and $\checkmark$ is a binary relation over its positions $\{1, \ldots, |c|\}$. The relation $\checkmark$ is used to encode intermediate states of for-loops. Intuitively, a process at position i performing a for-loop puts (i, j) into $\checkmark$ to mark that it has processed the position j.

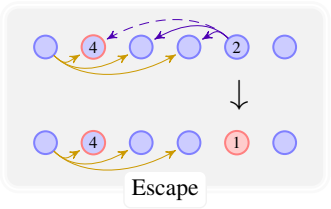
Formally, a parameterized system $\mathcal{P} = (Q, \Delta)$ which includes for-loop rules induces a transition system $\mathcal{T} = (C, \to)$ where $C \subseteq Q^+ \times (\mathbb{N} \times \mathbb{N})$. For technical convenience, we assume that a source of a for-loop rule in Δ is not a source of any other rule in Δ.[1] Then every for-loop rule **if foreach** $j \circ i : S$ **then** $q \to r$ **else** $q \to s$ induces transitions $t = (w, \checkmark) \to (w', \checkmark')$ with $w[i] = q$ for some $i : 1 \leq i \leq |w|$ which may be of the following three forms: (illustrated using the aforementioned example rule from Burn's protocol).

[1] Without this restriction, the state of a process would have to contain additional information recording which for-loop is the process currently performing. Note that the restriction does not limit the modeling power of the formalism. Any potential branching may be moved to predecessors of the sources of the for-loop.

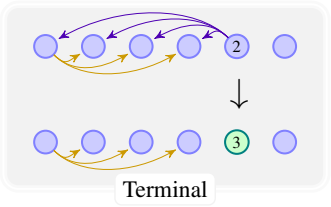
Iteration: The ith process checks that the state of a next unchecked process in the range is in S and marks it. That is, there is $j : 1 \leq j \leq |w|$ with $j \circ i$, $(i,j) \notin \checkmark$, $w[j] \in S$, and the resulting configuration has $w' = w$ and $\checkmark' = \checkmark \cup \{(i,j)\}$.



Escape: If the state of some process in the range which is still to be checked violates the loop condition, then the ith process may escape to the state s. That is, there is $j : 1 \leq j \leq |w|$ with $j \circ i$, $(i,j) \notin \checkmark$, and $w[j] \notin S$. The resulting configuration has $w'[k] = w[k]$ for all $k \neq i$ and $w[i] = s$. The execution of the for-loop ends and the marks of process i are reset, i.e., $\checkmark' = \checkmark \setminus \{(i,k) \mid k \in \mathbb{N}\}$.



Terminal: When the states of all processes from the range have been successfully checked, the for-loop ends and the ith process moves to the terminal state r. That is, if there is no $j : 1 \leq j \leq |w|$ with $j \circ i$ and $(i,j) \notin \checkmark$, then $w'[k] = w[k]$ for all $k \neq i$, $w'[i] = r$, and $\checkmark' = \checkmark \setminus \{(i,k) \mid k \in \mathbb{N}\}$.



Other rules behave as before on the w part of configurations and they do not influence the $\checkmark$ part. That is, a local, broadcast, or rendez-vous rule induces transitions $(w, \checkmark) \to (w', \checkmark)$ where $w \to w'$ is a transition induced by the rule as described in Section 2.

Verification. To verify systems with for-loop rules using our method, we define an abstraction α_k. Intuitively, we view a configuration $c = (w, \checkmark)$ as a graph with vertices being the positions of w and edges being defined by (i) the ordering of the positions and (ii) the relation $\checkmark$. The vertices are labeled by the states of processes at the positions. $\alpha_k(c)$ then returns the set of subgraphs of c where every subgraph contains a subset of at most k vertices of c (positions of w) and the maximal subset of edges of c adjacent with the chosen vertices.

Formally, given a configuration $c = (w, \checkmark)$, $\alpha_k(c)$ is the set of views $v = (w', \checkmark') \in C$ of size at most k (i.e., $|w'| = l \leq k$) such that there exists an injection $\rho : \{1, \ldots l\} \to \{1, \ldots, |w|\}, l \leq k$ where for all $i, j : 1 \leq i, j \leq l$:

- $i < j$ iff $\rho(i) < \rho(j)$,
- $w'[i] = w[\rho(i)]$ (i.e., $w' \sqsubseteq w$), and
- $(i,j) \in \checkmark'$ iff $(\rho(i), \rho(j)) \in \checkmark$.

The notions of concretization and abstract post-image are defined in the same manner as in Section 3 based on based on α. Lemma 1 holds here in the same wording (as shown in the appendix). Thus the verification method for systems with for-loops is analogous to the method of Section 3.

4.4 Tree Topology

We extend our method to systems where configurations are trees. For simplicity, we restrict ourselves to complete binary trees.

Trees. Let N be a prefix closed set of words over the alphabet $\{0,1\}$ called *nodes* and let Q be a finite set. A (binary) *tree* over Q is a mapping $t : N \to Q$. The node ε is called the *root*, nodes that are not prefixes of other nodes are called *leaves*. For a node $v = v'i, i \in \{0,1\}$, v' is the *parent* of v, the node $v0$ is the *left child* of v and $v1$ is its *right child*. Every node $v' = vw, w \in \{0,1\}^+$ is a *descendant* of v. The *depth* of the tree is the length of the longest leaf. A tree is *complete* if all its leaves have the same length and every non-leaf node has both children. A tree $t' : N' \to Q$ is a *subtree* of t, denoted $t' \preceq t$, iff there exists a injective map $e : N' \to N$ which respects the descendant relation and labeling. That is, $t'(v) = t(e(v))$ and v is a descendant of v' iff $e(v)$ is a descendant of $e(v')$.

Parameterized systems with tree topology. The definitions for parameterized systems with a tree topology are analogous to the definitions for systems with a linear topology (Section 2). A parameterized system $\mathcal{P} = (Q, \Delta)$ induces a transition system $\mathcal{T} = (C, \to)$ where C is the set of complete trees over Q. The set Δ of transition rules is a set of *local* and *tree* transition rules. The transitions of $\to$ are obtained from rules of Δ as follows. A *local* rule is of the form $s \to s'$ and it locally changes the label of a node from s to s'. A *tree* rule is a triple $s(s_0, s_1) \to s'(s'_0, s'_1)$. The rule can be applied to a node v and its left and right children v_0, v_1 with labels s, s_0, and s_1, respectively, and it changes their labels to s', s'_0, and s'_1, respectively.

The reachability problem is defined in a similar manner to the case of linear systems. The set B of minimal bad configurations is a finite set of trees over Q, I is a regular tree-language, and *Bad* is the upward closure of B w.r.t. the subtree relation $\preceq$. In the notation C_n and $\mathcal{R}_n$, n refers to the depth of trees rather than to the length of words.

Verification. The verification method of Section 3 is easily extended to the tree topology. The text of Section 3 can be taken almost verbatim with the difference that instead of words, we manipulate complete trees, subword relation is replaced by subtree relation, and k now refers to the depth of trees rather than the length of words. That is, a view of size k is a tree of depth k and the abstraction $\alpha_k(t)$ returns all complete subtrees of depth at most k of the tree t. Concretization and abstract post-image are defined analogously as in Section 3, based on α_k. The set I may be given in the form of a tree automaton. The computation of $\alpha_k(I)$ may be then done over the structure of the tree automaton. We can compute the abstract post-image since Lemma 1 holds here in the same wording as in Section 3. The test $\gamma_k(V) \cap Bad = \emptyset$ is carried out in the same way as in Section 3 since *Bad* is an upward closure of a set B w.r.t. $\preceq$. The points 1-4 of Section 3 are thus satisfied and Algorithm 1 can be used as a verification procedure for systems with tree topology.

4.5 Ring Topology

The method can be extended also to systems with a ring topology. In a parameterized system with ring topology, processes are organized in a circular array and they synchronize by near-neighbor communication. We model system with a ring topology as systems with linear topology of Section 2, where a configuration $c \in Q^+$ is interpreted as a circular word. The set Δ may contain local and *near-neighbor* transition rules.

A near-neighbor rule is a pair $(s_1 \to s_1', s_2 \to s_2')$. It induces the transition $c \to c'$ of $\to$ if either $c = c_L\, s_1 s_2\, c_R$ and $c' = c_L\, s_1' s_2'\, c_R$ (i.e. the 2 processes are adjacent in the configuration c) or $c = s_2\, \bar{c}\, s_1$ and $c' = s_2'\, \bar{c}\, s_1'$ (i.e. the 2 processes are positioned at the end of the configuration c). The latter case covers the communication between the extremities since configurations encode circular words.

Verification. A word u is a *circular subword* of a word v, denoted $u \trianglelefteq v$, iff there are v_1, v_2 such that $v = v_1 v_2$ and $u \sqsubseteq v_2\, v_1$. The only difference compared to the method for the systems with a linear topology is that the standard subword relation is in all definitions replaced by the circular subword relation $\trianglelefteq$. An equivalent of Lemma 1 holds here in unchanged wording, points 1-4 are satisfied, and Algorithm 1 is thus a verification procedure for systems with ring topology.

4.6 Multiset Topology

Systems which we refer to as systems with multiset topology are a special case of the systems with a linear topology of Section 2. Typical representatives of these systems are Petri nets, which correspond precisely to systems of Section 4 with only (generalized) rendez-vous transitions. Systems with multiset topology may contain all types of transitions including local, global, broadcast, and rendez-vous, with the exception of global transitions with the scope of indices $j > i$ and $j < i$ (i.e., only $j \neq i$ is permitted). Since the processes have no way of distinguishing their respective positions within a configuration, the notion of ordering of positions within a configuration is not meaningful and configurations can be represented as multisets.

5 Completeness for Well Quasi-Ordered Systems

In this section, will show that the scheme desribed by Algorithm 1 is complete for a wide class of well-quasi ordered systems. To state the result in general terms, we will first give some definitions from the theory of well quasi-ordered systems (c.f. [1]).

A *well quasi-ordering* (WQO) is a preorder $\preceq$ over a set S such that for every infinite sequence $s_1, s_2, \ldots$ of elements of S, there exists i and j such that $i < j$ and $s_i \preceq s_j$. The *upward-closure* $\uparrow T$ of a set $T \subseteq S$ w.r.t. $\preceq$ is the set $\{s \in S \mid \exists t \in T : t \preceq s\}$ and its *downward-closure* is the set $\downarrow T = \{s \in S \mid \exists t \in T : s \preceq t\}$. A set is *upward-closed* if it equals its upward-closure and it is *downward-closed* if it equals its downward-closure. If T is upward closed, its complement $S \setminus T$ is downward closed and, conversely, if T is downward closed, its complement is upward closed. For every upward closed set T, there exists a minimal (w.r.t $\subseteq$) set Gen such that $\uparrow Gen = T$, called *generator* of T, which is finite. If moreover $\preceq$ is a partial order, then Gen is unique.

A relation $R \subseteq S \times S$ is *monotonic* w.r.t. $\preceq$ if whenever $(s_1, s_2) \in R$ and $s_1 \preceq s_1'$, then there is s_2' with $(s_1', s_2') \in R$ and $s_2 \preceq s_2'$. Given a relation $f \subseteq S \times S$ monotonic w.r.t. $\preceq$ and a set $T \subseteq S$, it holds that if $f(T) \subseteq T$, then $f(\downarrow T) \subseteq \downarrow T$, where $f(T)$ is the image of T defined as $\{t' \mid \exists t \in T : (t, t') \in f\}$.

The reasoning in Section 3 is based on the natural notion of a size of a configuration. Its generalization is the notion of a *discrete measure* over a set S, a function $|.| : S \to \mathbb{N}$

which fulfills the property that for every $k \in \mathbb{N}$, $\{s \in S \mid |s| = k\}$ is finite. A discrete measure is necessary to obtain the completeness result as it allows enumerating elements of S of the same size. In particular, this property guarantees termination of the fixpoint computation on Line 3 of Algorithm 1. We note that the existence of a discrete measure is implied by a stronger restriction of [8] to the so called discrete transition systems.

We say that a transition system $\mathcal{T} = (C, \rightarrow)$ is *well-quasi ordered* by a WQO $\preceq \subseteq C \times C$ if $\rightarrow$ is monotonic w.r.t. $\preceq$. Given a well-quasi ordered transition system and a measure $|.| : C \rightarrow \mathbb{N}$, we define an abstraction function $\alpha_k, k \in \mathbb{N}$ such that $\alpha_k(c) = \{c' \in C \mid c' \preceq c\}$. The corresponding concretization γ_k and abstract post-image $Apost_k$ are then defined based on α_k and $|.|$ as in Section 3.1.

Lemma 2 holds here in the same wording as in Section 3. The main component of the completeness result is the following theorem.

Theorem 1. *Let $\mathcal{T} = (C, \rightarrow)$ be a well-quasi ordered transition system with a measure $|.|$. Let I be any subset of C and let Bad be upward-closed w.r.t. $\preceq$. Then, if $\mathcal{T}$ is safe w.r.t. I and Bad, then there is $k \in \mathbb{N}$ such that for $V = \mu X.\alpha_k(I) \cup Apost_k(X)$, $Bad \cap \gamma_k(V) = \emptyset$.*

Proof. Recall first that $\gamma_k, post, Apost_k, \alpha_k$ are monotonic functions w.r.t. $\subseteq$ for all $k \in \mathbb{N}$. Let Gen be the minimal generator of the upward closed set $C \setminus \downarrow\mathcal{R}$. We will prove that k can be chosen as $k = \max\{|c| \mid c \in Gen\}$. Such k exists because Gen is finite.

We first show an auxiliary claim that $\gamma_k(\alpha_k(\downarrow\mathcal{R})) \subseteq \downarrow\mathcal{R}$. Let $s \in \gamma_k(\alpha_k(\downarrow\mathcal{R}))$. For the sake of contradiction, suppose that $s \notin \downarrow\mathcal{R}$. We have that $s \in C \setminus \downarrow\mathcal{R} = \uparrow Gen$ and there is a generator $t \in Gen$ with $t \preceq s$. By the definition of k, $|t| \leq k$. Since $t \in Gen$, $t \notin \downarrow\mathcal{R}$ and hence $t \notin \alpha_k(\downarrow\mathcal{R})$. But due to this and since $t \preceq s$, we have that $s \notin \gamma_k(\alpha_k(\downarrow\mathcal{R}))$ (by the definition of γ_k) which contradicts the initial assumption and the claim is proven.

Next, we argue that $\alpha_k(\downarrow\mathcal{R})$ is stable under abstract post, that is, $Apost_k(\alpha_k(\downarrow\mathcal{R})) \subseteq \alpha_k(\downarrow\mathcal{R})$. Since $\mathcal{R}$ is stable under $post$ and $post$ is monotonic w.r.t. $\preceq$, we know that $\downarrow\mathcal{R}$ is stable under $post$ (that is, $post(\downarrow\mathcal{R}) \subseteq \downarrow\mathcal{R}$). Then, by the definition of $Apost_k$, and by monotonicity of α_k w.r.t. $\subseteq$, we have $Apost_k(\alpha_k(\downarrow\mathcal{R})) = \alpha_k(post(\gamma_k(\alpha_k(\downarrow\mathcal{R})))) \subseteq \alpha_k(post(\downarrow\mathcal{R})) \subseteq \alpha_k(\downarrow\mathcal{R})$.

Since $\downarrow\mathcal{R}$ contains I, $\alpha_k(I) \subseteq \alpha_k(\downarrow\mathcal{R})$. $\alpha_k(\downarrow\mathcal{R})$ is thus a fixpoint of $\lambda X.\alpha_k(I) \cup Apost_k(X)$. Because V is the least fixpoint of $\lambda X.\alpha_k(I) \cup Apost_k(X)$, $V \subseteq \alpha_k(\downarrow\mathcal{R})$. From, $\mathcal{R} \cap Bad = \emptyset$ and since Bad is upward closed, we know that $\downarrow\mathcal{R} \cap Bad = \emptyset$. Because $\gamma_k(V) \subseteq \gamma_k(\alpha_k(\downarrow\mathcal{R})) \subseteq \downarrow\mathcal{R}$ and $\downarrow\mathcal{R} \cap Bad = \emptyset$, $\gamma_k(V) \cap Bad = \emptyset$. □

Theorem 1 guarantees that for a safe well quasi-ordered system, there exists k for which the test on line 4 of Algorithm 1 succeeds. Conversely, Lemma 2, which, as mentioned above, still holds for the general class of well-quasi ordered systems, then assures than if the test on line 2 succeeds, the system is indeed safe.

Complete algorithm. The schema described by Algorithm 1 (or its variant from Section 4.2 if the transition relation is not size-preserving) gives a complete verification procedure for a well quasi-ordered system provided that all the four steps of its for-loop can be effectively evaluated. This is guaranteed by the following requirements:

i. $\alpha_k(I)$ can be computed,
ii. the measure $|.|$ is discrete,

iii. for a configuration c, $post(c)$ and $\alpha_k(c)$ can be computed,
iv. for a finite set of views V, $\gamma_k^{k+1}(V)$ can be computed, and
v. a variant of Lemma 1 holds.

Point (i) is point 1 of Section 3. Points (ii)-(v) guarantee that we can compute abstract post-image (point 2 of Section 3). We can test $\gamma_k(V) \cap Bad = \emptyset$ (point 3 of Section 3) since due to (ii), V is always finite. Exact reachability analysis of configurations of a bounded size (point 4 of Section 3) can be carried out since we can iterate *post* due to (iii) and the iteration terminates after a finite number of steps due to (ii). Point (ii) also assures termination of the computation of the fixpoint on line 3 (V is always finite).

Overall, Algorithm 1 is a complete verification procedure for parameterized systems of Section 2 with local and existential transitions rules, broadcast and rendez-vous. The induced transition relation is indeed monotonic w.r.t. the preorder $\sqsubseteq$ which is a WQO and the length of a configuration is a discrete measure. An important subclass of such systems are Petri nets, which, as mentioned in Section 4, correspond to systems with multiset topology and generalized rendez-vous transition rules. Systems of Section 2 with universally guarded transition rules do not satisfy the assumptions: the induced transition relation is not monotonic.

6 Experimental Results

Based on our method, we have implemented a prototype in OCaml to check safety properties for a number of parameterized systems with different topologies. The examples cover cache coherence protocols, communication protocols through trees and rings and mutual exclusion protocols.

Table 1. Experimental Results

	Protocol	**Time**	k	$\lvert V\rvert$	$\gamma_k^{k+\ell}(V)$
Array	Demo (toy example)	0.01s	2	17	53
	Burns	0.01s	2	34	186
	Dijkstra	0.07s	2	93	695
	Szymanski	0.02s	2	48	264
Multiset	MOSI Coherency	0.01s	1	10	23
	German's Coherency	15.3s	6	1890	15567
Petri Net	German (simplified)	0.03s	2	43	96
	BH250	2.85s	2	503	503
	MOESI Coherency	0.01s	1	13	20
	Critical Section	0.01s	5	27	46
	Kanban	?	≥ 20	?	?
Tree	Percolate	0.05s	2	34	933
	Tree Arbiter	0.7s	2	88	7680
	Leader Election	0.1s	2	74	362
Ring	Token Passing	0.01s	2	2	2

We report the results in Table 1, running on a 2.4 GHz laptop with 4GB memory. We have categorized the experiments per topology. We display the running times (in seconds), the value of k and the final number of views generated ($|V|$). In most cases, the method terminates almost immediately illustrating the *small model property*: all patterns occur for small instances of the system. Observe that the sizes of the views are small as well, confirming the intuition that interactions between processes are of limited scope.

The bulk of the algorithm lies in the computation of the set $\gamma_k^{k+\ell}(V)$ and also the set $\mathcal{R}_k$. An example on which the algorithm fails is the *Kanban* system from [24]. This is a typical case where the cut-off condition is satisfied at high values of k. [24] refers to the computation of, at least, the set $\mathcal{R}_{20}$. $\mathcal{R}_{20}$ is large and so is the concretization of its views.

7 Related Work

An extensive amount of work has been devoted to regular model checking, e.g. [25,12]; and in particular augmenting regular model checking with techniques such as widening [9,32], abstraction [10], and acceleration [5]. All these works rely on computing the transitive closure of transducers or on iterating them on regular languages. Our method is significantly simpler and more efficient.

A technique of particular interest for parameterized systems is that of *counter abstraction*. The idea is to keep track of the number of processes which satisfy a certain property [22,17,13,14,30]. In general, counter abstraction is designed for systems with unstructured or clique architectures. As mentioned, our method can cope with these kinds of systems but also with more general classes of topologies. Several works reduce parameterized verification to the verification of finite-state models. Among these, the *invisible invariants* method [6,31] and the work of [29] exploit cut-off properties to check invariants for mutual exclusion protocols. The success of the method depends on the heuristic used in the generation of the candidate invariant. This method sometimes (e.g. for German's protocol) requires insertion of auxiliary program variables for completing the proof. The nature of invariants generated by our method is similar to that of the aforementioned works, since our invariant sets of views of size at most k can be seen as universally quantified assertions over reachable k-tuples of processes.

In [7], finite-state abstractions for verification of systems specified in WS1S are computed on-the-fly by using the weakest precondition operator. The method requires the user to provide a set of predicates on which to compute the abstract model.

The idea of refining the view abstraction by increasing k is similar in spirit to the work of [28] which discusses increasing precision of thread modular verification (Cartesian abstraction) by remembering some relationships between states of processes. Their refinement mechanism is more local, targeting the source of undesirable imprecision; however, it is not directly applicable to parameterized verification.

Environment abstraction [11] combines predicate abstraction with the counter abstraction. The technique is applied to Szymanski's algorithm. The model of [11] contains a more restricted form of global conditions than ours, and also does not include

features such as broadcast communication, rendez-vous communication, and dynamic creation and deletion of processes.

Recently, we have introduced the method of *monotonic abstraction* [3] that combines regular model checking with abstraction in order to produce systems that have monotonic behaviors w.r.t. a well quasi-ordering on the state space. In contrast to the method of this paper, the abstract system still needs to be analyzed using full symbolic reachability analysis on an infinite-state system. The only work we are aware of which attempts to automatically verify systems with non-atomic global transitions is [4] which applies monotonic abstraction. The abstraction in this case amounts to a verification procedure that operates on unbounded graphs, and thus is a non-trivial extension of the existing framework. As we saw, our method is easily extended to the case of non-atomic transitions.

The method of [21,20] and its reformulated, generic version of [19] are in principle similar to ours. They come with a complete method for well-quasi ordered systems which is an alternative to backward reachability analysis based on a forward exploration. Unlike our method, they target well-quasi ordered systems only and have not been instantiated for topologies other than multisets and lossy channel systems.

Constant-size cut-offs have been defined for ring networks in [16] where communication is only allowed through token passing. More general communication mechanisms such as guards over local and shared variables are described in [15]. However, the cut-offs are linear in the number of states of the components, which makes the verification task intractable on most of our examples.

The closest work to ours is the one in [24] that also relies on dynamic detection of cut-off points. The class of systems considered in [24] corresponds essentially to Petri nets. In particular, it cannot deal with systems with linear or tree-like topologies. The method relies on the ability to perform backward reachability analysis on the underlying transition system. This means that the algorithm of [24] cannot be applied on systems with undecidable reachability problems (such as the ones we consider in this paper). The method of [24] is yet complete.

8 Conclusion and Future Work

We have presented a uniform framework for automatic verification of different classes of parameterized systems with topologies such as words, trees, rings, or multisets, with an extension to handle non-atomic global conditions. The framework allows to perform parameterized verification by only considering a small set of instances of the system. We have proved that the presented algorithm is complete for a wide class of well quasi-ordered systems. Based on the method, we have implemented a prototype which performs efficiently on a wide range of benchmarks.

We are currently working on extending the framework to the case of multi-threaded programs operating on dynamic heap structures. These systems have notoriously complicated behaviors. Showing that verification can be carried out through the analysis of only a small number of threads would allow for more efficient algorithms for these systems. Furthermore, our algorithm relies on a very simple abstraction function, where a configuration of the system is approximated by its sub-structures (subwords, subtrees,

etc.). We believe that our approach can be lifted to more general classes of abstractions. This would allow for abstraction schemes that are more precise than existing ones, e.g., thread-modular abstraction [18] and Cartesian abstraction [27].

Obviously, the bottleneck in the application of the method is when the cut-off condition is only satisfied at high values of k (see e.g., the *Kanban* example in Section 6). We plan therefore to integrate the method with advanced tools that can perform efficient forward reachability analysis, like SPIN [23], and to use efficient symbolic encodings for compact representations for the set of views.

Acknowledgements. This work was supported by the Uppsala Programming for Multicore Architectures Research Center (UpMarc) and the Czech Science Foundation (project P103/10/0306).

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bulletin of Symbolic Logic 16(4), 457–515 (2010)
2. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321 (1996)
3. Abdulla, P.A., Delzanno, G., Ben Henda, N., Rezine, A.: Regular Model Checking Without Transducers (On Efficient Verification of Parameterized Systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
4. Abdulla, P.A., Ben Henda, N., Delzanno, G., Rezine, A.: Handling Parameterized Systems with Non-atomic Global Conditions. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 22–36. Springer, Heidelberg (2008)
5. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Regular Model Checking Made Simple and Efficient. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 116–130. Springer, Heidelberg (2002)
6. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.: Parameterized Verification with Automatically Computed Inductive Assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
7. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 317–330. Springer, Heidelberg (2002)
8. Bingham, J.D., Hu, A.J.: Empirically Efficient Verification for a Class of Infinite-State Systems. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 77–92. Springer, Heidelberg (2005)
9. Boigelot, B., Legay, A., Wolper, P.: Iterating Transducers in the Large. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
10. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract Regular Model Checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
11. Clarke, E., Talupur, M., Veith, H.: Environment Abstraction for Parameterized Verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2006)
12. Dams, D., Lakhnech, Y., Steffen, M.: Iterating Transducers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 286–297. Springer, Heidelberg (2001)
13. Delzanno, G.: Automatic Verification of Cache Coherence Protocols. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)

14. Delzanno, G.: Verification of consistency protocols via infinite-state symbolic model checking. In: FORTE 2000. IFIP Conference Proceedings, vol. 183, pp. 171–186. Kluwer (2000)
15. Emerson, E.A., Kahlon, V.: Reducing Model Checking of the Many to the Few. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000)
16. Emerson, E.A., Namjoshi, K.: Reasoning about rings. In: POPL 1995, pp. 85–94 (1995)
17. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999. IEEE Computer Society (1999)
18. Flanagan, C., Qadeer, S.: Thread-Modular Model Checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
19. Ganty, P., Raskin, J.-F., Van Begin, L.: A Complete Abstract Interpretation Framework for Coverability Properties of WSTS. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 49–64. Springer, Heidelberg (2006)
20. Geeraerts, G., Raskin, J.-F., Van Begin, L.: Expand, Enlarge and Check.. Made Efficient. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 394–407. Springer, Heidelberg (2005)
21. Geeraerts, G., Raskin, J.F., Begin, L.V.: Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. J. Comput. Syst. Sci. 72(1), 180–203 (2006)
22. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
23. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
24. Kaiser, A., Kroening, D., Wahl, T.: Dynamic Cutoff Detection in Parameterized Concurrent Programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010)
25. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theor. Comput. Sci. 256, 93–112 (2001)
26. Lynch, N.A., Shamir, B.P.: Distributed algorithms, lecture notes for 6.852, fall 1992. Tech. Rep. MIT/LCS/RSS-20, MIT (1993)
27. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-Modular Verification Is Cartesian Abstract Interpretation. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 183–197. Springer, Heidelberg (2006)
28. Malkis, A., Podelski, A., Rybalchenko, A.: Precise Thread-Modular Verification. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 218–232. Springer, Heidelberg (2007)
29. Namjoshi, K.S.: Symmetry and Completeness in the Analysis of Parameterized Systems. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 299–313. Springer, Heidelberg (2007)
30. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with $(0, 1, \infty)$-Counter Abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)
31. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic Deductive Verification with Invisible Invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
32. Touili, T.: Regular Model Checking using Widening Techniques. Electronic Notes in Theoretical Computer Science 50(4) (2001); Proc. of VEPAS 2001

Int J Softw Tools Technol Transfer (2017) 19:549–563
DOI 10.1007/s10009-016-0415-4

TACAS 2013

An integrated specification and verification technique for highly concurrent data structures for highly concurrent data structures

Parosh Aziz Abdulla[1] · **Frédéric Haziza**[1] · **Lukáš Holík**[1,2] · **Bengt Jonsson**[1] · **Ahmed Rezine**[3]

Published online: 16 March 2016

Abstract We present a technique for automatically verifying safety properties of concurrent programs, in particular programs that rely on subtle dependencies of local states of different threads, such as lock-free implementations of stacks and queues in an environment without garbage collection. Our technique addresses the joint challenges of infinite-state specifications, an unbounded number of threads, and an unbounded heap managed by explicit memory allocation. Our technique builds on the automata-theoretic approach to model checking, in which a specification is given by an automaton that observes the execution of a program and accepts executions that violate the intended specification. We extend this approach by allowing specifications to be given by a class of infinite-state automata. We show how such automata can be used to specify queues, stacks, and other data structures, by extending a data-independence argument. For verification, we develop a shape analysis, which tracks correlations between pairs of threads, and a novel abstraction to make the analysis practical. We have implemented our method and used it to verify programs, some of which have not been verified by any other automatic method before.

F. Haziza and L. Holik were in part supported by the Uppsala Programming for Multicore Architectures Research Center (UPMARC). L. Holik was in part supported by the Czech Science Foundation (project 13-37876P), the internal projects of Brno University of Technology FIT-S-12-1 and FIT-S-14-2486. A. Rezine was in part supported by the CENIIT research organization at Linköping (project 12.04).

Ahmed Rezine
ahmed.rezine@liu.se

[1] Uppsala University, Uppsala, Sweden

[2] Brno University of Technology, Brno, Czech Republic

[3] Linköping University, Linköping, Sweden

Keywords Verification · Pointer programs · Explicit memory allocation · Queue · Stack · Unbounded · Concurrency · Specification · Linearizability

1 Introduction

We consider one of the most difficult current challenges in software verification, namely to automate its application to algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared state. Such algorithms are of central importance in concurrent programs. They are widely used in libraries, such as the Intel Threading Building Blocks or the java.util.concurrent package, to provide efficient concurrent realizations of simple interface abstractions. They are notoriously difficult to get correct and verify, since they often employ fine-grained synchronization and avoid locking wherever possible. A number of bugs in published algorithms have been reported [11,20]. It is therefore important to develop efficient techniques for verifying conformance to simple abstract specifications of overall functionality, a concurrent implementation of a common data type abstraction, such as a queue, should be verified to conform to a simple abstract specification of a (sequential) queue.

We present an integrated technique for specifying and automatically verifying whether a concurrent program conforms to an abstract specification of its functionality. Our starting point is the automata-theoretic approach to model checking [31], in which programs are specified by automata that accept precisely those executions that violate the intended specification, and verified by showing that these automata never accept when they are composed with the program. This approach is one of the most successful approaches to automated verification of finite-state programs, but is still

insufficiently developed for infinite-state programs. In order to use this approach for our purposes, we must address a number of challenges.

1. The abstract specification is infinite-state, because the implemented data structure may contain an unbounded number of data values from an infinite domain.
2. The program is infinite-state in several dimensions: it (i) consists of an unbounded number of concurrent threads, (ii) uses unbounded dynamically allocated memory, and (iii) the domain of data values is unbounded.
3. The program does not rely on automatic garbage collection, but manages memory explicitly. This requires additional mechanisms to avoid the ABA problem, i.e., that a thread mistakenly confuses an outdated pointer with a valid one. We assume a sequentially consistent memory model.

Each of these challenges requires a significant advancement over current specification and verification techniques.

We cope with challenge 1 by combining two ideas. First, we present a novel technique for specifying programs by a class of automata, called *observers*. They extend automata, as used by [31], by being parameterized on a finite set of variables that assume values from an unbounded domain. This allows to specify properties that should hold for an infinite number of data values. In order to use our observers to specify queues, stacks, etc., where one must "count" the number of copies of a data value that have been inserted but not removed, we must extend the power of observers by a second idea. This is a data independence argument, adapted from Wolper [35], which implies that it is sufficient to consider executions in which any data value is inserted at most once. This allows us to succinctly specify data structures such as queues and stacks, using observers with typically less than 3 variables.

To cope with challenge 2(i), we would like to adapt the successful thread-modular approach [5], which verifies a concurrent program by generating an invariant that correlates the global state with the local state of an arbitrary thread. However, to cope with challenge 3, the generated invariant must be able to express that *at most* one thread accesses some cell on the global heap. Since this cannot be expressed in the thread-modular approach, we therefore extend this approach to generate invariants that correlate the global state with the local states of an arbitrary *pair* of threads.

To cope with challenge 2(ii) we need to use shape analysis. We adapt a variant of the transitive closure logic by Bingham and Rakamarić [6] for reasoning about heaps with single selectors, to our framework. This formalism tracks reachability properties between pairs of pointer variables, and we adapt it to our analysis, in which pairs of threads are correlated. On top of this, we have developed a novel optimization, based on the observation that it suffices to track the possible relations between each pair of pointer variables separately (in other words, to use Cartesian abstraction over conjunctions of relations between variables), analogously to the use of DBMs used in reasoning about timed automata [10]. Finally, we cope with challenge 2(iii) by first observing that data values are compared only by equalities or inequalities, and then employing suitable standard abstractions on the concerned data domains.

To handle challenge 3, we augment our abstract domain to allow it to capture the ordering between the counters used to track the relative ages of the pointers.

We have implemented our technique, and applied it to specify and automatically verify that a number of concurrent programs are linearizable implementations of stacks and queues [17]. This shows that our new contributions result in an integrated technique that addresses the challenges 1–3, and can fully automatically verify a range of concurrent implementations of common data structures. In particular, our approach advances the power of automated verification in the following ways.

- We present a direct approach for verifying that a concurrent program is a linearizable implementation of, e.g., a queue, which consists in checking a few small properties of the algorithm, and is thus suitable for automated verification. Previous approaches typically verified linearizability separately from conformance to a simple abstraction, most often using simulation-based arguments, which are harder to automate than simple property-checking.
- We can automatically verify concurrent programs that use explicit memory management. This was previously beyond the reach of automatic methods.

In addition, on examples that have been verified automatically by previous approaches, our implementation is in many cases significantly faster.

Overview We give an overview of how our technique can be used to show that a concurrent program is a linearizable implementation of a data structure. As described in Sect. 2, we consider concurrent programs consisting of an arbitrary number of sequential threads that access shared global variables and a shared heap using a finite set of methods. Linearizability provides the illusion that each method invocation takes effect instantaneously at some point (called the linearization point) between method invocation and return [17]. In Sect. 3, we show how to specify this correctness condition by first instrumenting each method to generate a so-called abstract event whenever a linearization point is passed. We also introduce *observers*, and show how to use them for specifying properties of sequences of abstract events. In Sect. 4, we introduce the data independence argument that allows observers to specify queues, stacks, and other data struc-

tures over unbounded data domains as presented in Sect. 5. In Sect. 6, we describe our analysis for checking that the cross-product of the program and the observer cannot reach an accepting location of the observer. The analysis is based on a shape analysis, which generates an invariant that correlates the global state with the local states of an arbitrary pair of threads. We also introduce an optimization that tracks the possible relations between each pair of pointer variables separately. We report on experimental results in Sect. 7. Section 8 contains conclusions and directions for future work. This article details the approach presented in [3] and includes the correctness proofs of the stack and queue data-structures.

Related work Much previous work on verification of concurrent programs has concerned the desection of generic concurrency problems, such as race conditions, atomicity violations, or deadlocks [15,23,24]. Verification of conformance to a simple abstract specification has been performed using refinement techniques, which establish simulation relations between the implementation and specification, using partly manual techniques [9,12,13,34].

Amit et al. [4] verify linearizability by verifying conformance to an abstract specification, which is the same as the implementation, but restricted to serialized executions. They build a specialized abstract domain that correlates the state (including the heap cells) of a concrete thread and the state of the serialized version, and a sequential reference data structure. The approach can handle a bounded number of threads. Berdine et al. [5] then generalize the approach of to an unbounded number of threads by making the shape analysis thread-modular. In our approach, we need not keep track of heaps emanating from sequential reference executions, and so we can use a simpler shape analysis. Plain thread-modular analysis is also not powerful enough to analyze, e.g., algorithms with explicit memory management. The technique in [5] thus improves the precision by correlating local states of different threads. This causes, however, a severe state-space explosion, which limits the applicability of the method.

Vafeiadis [28] formulates the specification using an unbounded sequence of data values that represent, e.g., a queue or a stack. He verifies conformance using a specialized abstraction to track values in the queue and correlate them with values in the implementation. Like [26], our technique for handling values in queues need only consider a small number of data values (not an unbounded one), for which it is sufficient to track equalities. The approach is extended in [29] to automatically infer the position of linearization points: these have to be supplied in our approach.

Our use of data variables in observers for specifying properties that hold for all data values in some domain is related in spirit to the identification of arbitrary but fixed objects or resources by Emmi et al. [14] and Kidd et al. [19]. In the framework of regular model checking, universally quantified temporal logic properties can be compiled into automata with data variables that are assigned arbitrary initial values [1].

Segalov et al. [25] continue the work of [5] by also considering an analysis that keeps track of correlations between threads. They strive to counter the state-space explosion that [5] suffers from, and propose optimizations that are based on the assumption that inter-process relationships that need to be recorded are relatively loose, allowing a rather crude abstraction over the state of one of the correlated threads. These optimizations do not work well when thread correlations are tight. Our experimental evaluation in Sect. 7 shows that our optimizations of the thread correlation approach achieve significantly better analysis times than [25].

There are several works that apply different verification techniques to programs with a bounded number of threads, including the use of TVLA [36]. Several approaches produce decidability results under limited conditions [8], or techniques based on non-exhaustive testing [7] or state-space exploration [33] for a bounded number of threads.

2 Programs

We consider systems consisting of an arbitrary number of concurrently executing threads. Each thread may at any time invoke one of a finite set of methods. Each method declares local variables (including the input parameters of the method) and a method body. In this paper, we assume that variables are either pointer variables (to heap cells), or data variables (assuming values from an unbounded or infinite domain, which will be denoted by $\mathbb{D}$). The body is built in the standard way from atomic commands using standard control flow constructs (sequential composition, and loop constructs). Method execution is terminated by executing a `return` command, which may return a value. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread that is invoking the corresponding method. We assume that the global variables and the heap are initialized by an initialization method, which is executed once at the beginning of program execution.

Atomic commands include assignments between data variables, pointer variables, or fields of memory cells pointed to by a pointer variable. The command **new**`node()` allocates a new structure of type `node` on the heap, and returns a reference to it. The cell is deallocated by the command **free**. The compare-and-swap command `CAS(&a,b,c)` atomically compares the values of `a` and `b`. If equal, it assigns the value of `c` to `a` and returns `TRUE`, otherwise, it leaves `a` unchanged and returns `FALSE`.

As an example, Fig. 1 shows a version of the concurrent queue by Michael and Scott [21]. The program represents a queue as a linked list from the node pointed to by `Head`

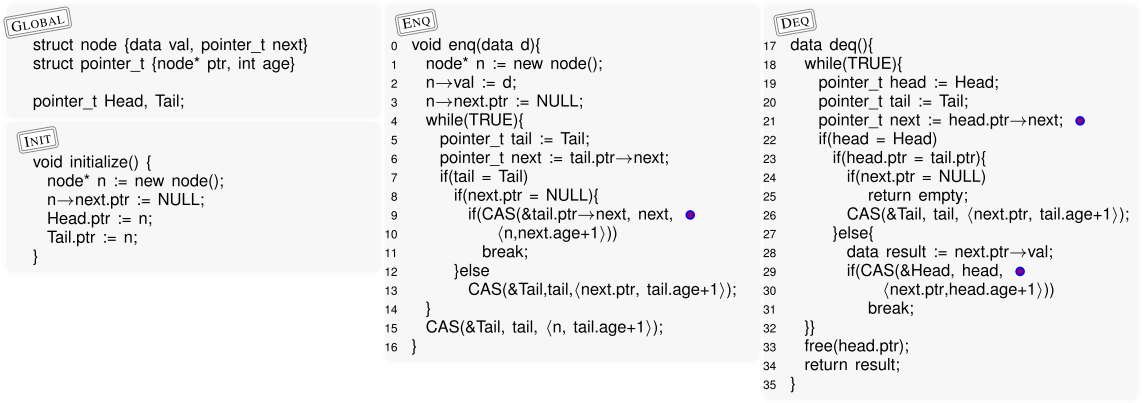


Fig. 1 Michael & Scott's non-blocking queue [21]

to a node that is either pointed by `Tail` or by `Tail`'s successor. The global variable `Head` always points to a dummy cell whose successor, if any, stores the head of the queue. In the absence of garbage collection, the program must handle the ABA problem where a thread mistakenly assumes that a globally accessible pointer has not been changed since it previously accessed that pointer. Each pointer is therefore equipped with an additional `age` field, which is incremented whenever the pointer is assigned a new value.

The queue can be accessed by an arbitrary number of threads, either by calling an enqueue method `enq(d)`, which inserts a cell containing the data value `d` at the tail, or by calling a dequeue method `deq(d)`, which returns `empty` if the queue is empty, and otherwise advances `Head`, deallocates the previous dummy cell and finally returns the data value stored in the new dummy cell. The algorithm uses the atomic compare-and-swap (CAS) operation. For example, the command `CAS(&Head, head, ⟨next.ptr,head.age+1⟩)` at line 29 of the `deq` method checks whether the extended pointer `Head` equals the extended pointer `head` (meaning that both fields must agree). If not, it returns `FALSE`. Otherwise it returns `TRUE` after assigning `⟨next.ptr,head.age+1⟩` to `Head`.

3 Specification by observers

To specify a correctness property, we instrument each method to generate abstract events. An *abstract event* is a term of the form $l(d_1, \ldots, d_n)$ where l is an event type, taken from a finite set of event types, and $d_1, \ldots, d_n$ are data values in $\mathbb{D}$. To specify linearizability, the abstract event $l(d_1, \ldots, d_n)$ generated by a method should be such that l is the name of the method, and $d_1, \ldots, d_n$ is the sequence of actual parameters and return values in the current invocation of the method. This can be established using standard sequential verification techniques.

We illustrate how to instrument the program of Fig. 1 in order to specify that it is a linearizable implementation of a queue. The linearization points ● are at lines 9, 21 and 29. For instance, line 9 of the `enq` method called with data value `d` is instrumented to generate the abstract event `enq(d)` when the `CAS` command succeeds; no abstract event is generated when the `CAS` fails. Generation of abstract events can be conditional. For instance, line 21 of the `deq` method is instrumented to generate `deq(empty)` when the value assigned to `next` satisfies `next.ptr = NULL` (i.e., it will cause the method to return `empty` at line 25).

Each execution of the instrumented program will generate a sequence of abstract events called a *trace*. A *correctness property* (or simply a *property*) is a set of traces. We say that an instrumented program *satisfies* a property if each trace of the program is in the property. In contrast to the classical (finite-state) automata-theoretic approach [31], we specify properties by *infinite-state* automata, called *observers*. An observer has a finite set of control locations, and a finite set of data variables that range over potentially infinite domains. It observes the trace and can reach an accepting control location if the trace is not in the property.

Formally, let a *parameterized event* be a term of the form $l(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are formal parameters. We will write $\overline{p}$ for $p_1, \ldots, p_n$, and $\overline{d}$ for $d_1, \ldots, d_n$. An *observer* consists of a finite set of *observer locations*, one of which is *initial* and some of which are *accepting*, a finite set of *observer variables*, and a finite set of *transitions*. Each transition is of form $s \xrightarrow{l(\overline{p});g} s'$ where s, s' are observer locations, $l(\overline{p})$ is a parameterized event, and the guard g is a Boolean combination of equalities over formal parameters $\overline{p}$, and observer variables. Intuitively, it denotes that the observer can move from location s to location s' when an abstract event of form $l(\overline{d})$ is generated such that $g[\overline{d}/\overline{p}]$ is true. Note that the values of observer variables are not updated in a transition. Note also that an observer need not be determin-

istic. An *observer configuration* is a pair $\langle s, \vartheta \rangle$, where s is an observer location, and ϑ maps each observer variable to a value in the data domain $\mathbb{D}$. The configuration is initial if s is initial; thus the variables can assume any initial values. An *observer step* is a triple $\langle s, \vartheta \rangle \xrightarrow{l(\overline{d})} \langle s', \vartheta \rangle$ such that there is a transition $s \xrightarrow{l(\overline{p});g} s'$ for which $g[\overline{d}/\overline{p}]$ is true. A *run* of the observer on a trace $\sigma = l_1(\overline{d}_1)l_2(\overline{d}_2)\cdots l_n(\overline{d}_n)$ is a sequence of observer steps $\langle s_0, \vartheta \rangle \xrightarrow{l_1(\overline{d}_1)} \cdots \xrightarrow{l_n(\overline{d}_n)} \langle s_n, \vartheta \rangle$ where s_0 is the initial observer location. The run is *accepting* if s_n is accepting. A trace σ is *accepted* by an observer $\mathcal{A}$ if $\mathcal{A}$ has an accepting run on σ. The property specified by $\mathcal{A}$ is the set of traces that are not accepted by $\mathcal{A}$.

Since the data variables can assume arbitrary initial values, observers can specify properties that are universally quantified over all data values. If a trace violates such a property for some data values, the observer can non-deterministically choose these as initial values of its variables, and thereafter detect the violation when observing the trace. Several data structures can be specified by a collection of properties, each of which is represented by an observer. Note that an observer is used to capture the behavior of a data structure by following the sequence of encountered events, irrespective of the actual implementation of the data structure.

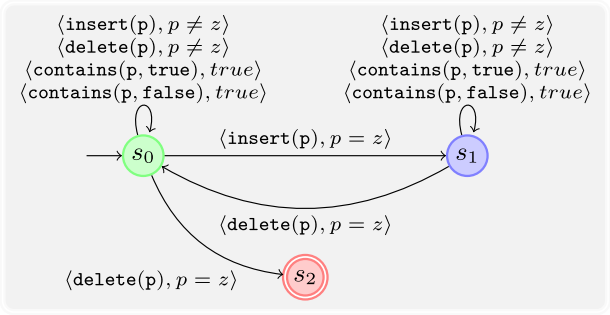


Fig. 2 An observer for deleting a non-present data value

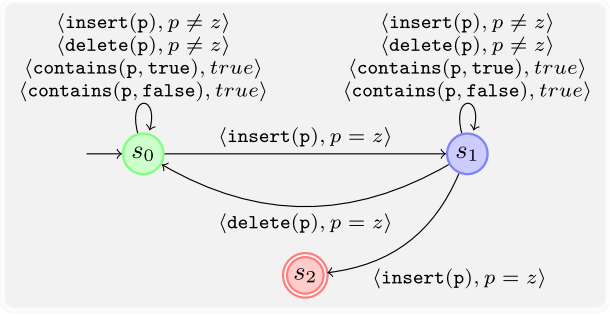


Fig. 3 An observer for inserting a data value that is already present

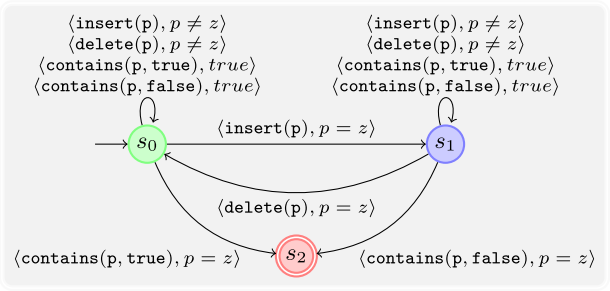


Fig. 4 An observer for observing a missing data value or for missing a present data value

3.1 Application: sets specification

We use three observers to capture all the set traces over the events in $\cup_{d\in\mathbb{D}}\{\texttt{contains(d, true)}, \texttt{contains(d, false)}\} \cup \{\texttt{insert(d)}, \texttt{delete(d)}\}$ that violate the expected behavior of a correct set implementation. The three observers in Figs. 2, 3 and 4 have three states s_0, s_1 and s_2. In these observers, the initial state s_0 corresponds to positions in the runs where the non-deterministically tracked value stored in the observer variable z is not present in the set (i.e. each time it has been inserted it got deleted afterwards). The state s_1 corresponds to positions in the runs where the tracked value is present in the set (i.e. it has not been deleted since it was last inserted). The accepting state s_2 corresponds to positions in the runs where the bad behavior captured by the respective observers has been observed. For the observer depicted in Fig. 2, the captured bad behaviors are those where a data value is deleted although it is not present in the set. For the observer of Fig. 3, the captured bad behaviors are those where a data value is inserted although it is already present in the set. For the observer of Fig. 4, the captured behaviors are those where a value erroneously appears to be contained or absent in the set. The three observers can be merged into a single observer.

3.2 Observers alone cannot "count"

In the previous paragraph, we showed how observers can specify behaviors of data structures such as sets. Registers and similar data structures (such as caches) where there is an a priori fixed bound on the number of equal data values that have been inserted but not yet retrieved can also be specified using appropriate observers. There are however data structures and properties where observers alone are not enough to capture the specification. Queues and stacks are examples of such data structures. Here, the difference between the number of times a data value may be inserted and the number of times it is retrieved can be arbitrary. In other words, the number of copies of the same data value that are present in the data structures can be arbitrary. As a result, one must be able

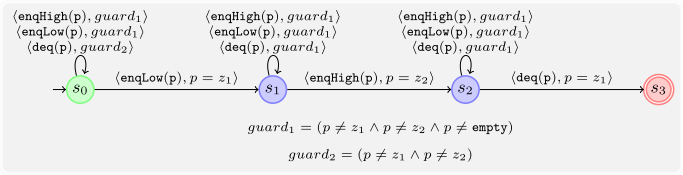


Fig. 5 A trace observer for checking that a low priority data value cannot be dequeued if there is a high priority data value that was later inserted. The variables z_1, z_2 are observer variables, and `empty` in an observer constant

to "count" the number of equal data values that have been inserted but not yet removed. Such data structures require therefore non-regular specifications in general. By restricting the allowed traces we can again use observers as defined in this section.

For instance, assume a queue where data values are assigned a low (respectively, high) priority each time they are inserted with `enqLow()` (respectively, `enqHigh()`). A correct implementation of such a priority queue will not return a data value with low priority if one with high priority was later inserted. The observer of Fig. 5 captures all traces that violate this property and where no data value `d` is enqueued twice (whether with `enqLow(d)`, `enqHigh(d)`, or both). In the following section, we build on the idea of specifying restricted traces using observers and show, by leveraging on a data independence argument, that this is sufficient to completely specify data structures such as stacks and queues.

4 Data independence

We adapt a data independence argument from Wolper [35]. The argument assumes that for each trace, there is a fixed subset of all occurrences of data values in the trace, called the set of *input occurrences*. Formally, this subset can be arbitrary, but to make the argument work, input occurrences should typically be the data values that are provided as actual parameters of method invocations. Thus, in the program of Fig. 1, the input occurrences are the parameters of `enq(d)` events, whereas parameters of `deq(d)` events are *not* input occurrences, since they are provided as return values.

Let us introduce some definitions. A trace is *differentiated* if all its input occurrences are pairwise different. A *renaming* is any function $f : \mathbb{D} \mapsto \mathbb{D}$ on the domain of data values. A renaming f can be applied to trace σ, resulting in the trace $f(\sigma)$, where each data value `d` in σ has been replaced by $f(\mathtt{d})$. A set Σ of traces is *data independent* if for every trace $\sigma \in \Sigma$ the following two conditions hold:

- $f(\sigma) \in \Sigma$ for every renaming f, and
- there exists a differentiated trace $\sigma_d \in \Sigma$ with $f(\sigma_d) = \sigma$ for some renaming f.

We say that a program is *data independent* if the set of its traces is data independent. A program, like the one in Fig. 1, can typically be shown to be data independent by a simple syntactic analysis that checks that data values are not manipulated or tested, but only copied. In a similar manner, a correctness property is *data independent* if the set of traces that it specifies is data independent. The following theorem states an important observation.

Theorem 1 *For every data-independent sets of traces Σ and Σ', $\Sigma \subseteq \Sigma'$ iff the differentiated traces of Σ are in Σ'.*

Proof If $\Sigma \subseteq \Sigma'$ then the differentiated traces of Σ are included in Σ'. Let σ be an arbitrary trace in Σ. We show $\sigma \in \Sigma'$. By data independence of Σ, there is a differentiated trace $\sigma_d \in \Sigma$ and a renaming f such that $f(\sigma_d) = \sigma$. By assumption, σ_d is also in Σ'. By data independence of Σ', $f(\sigma_d)$ is also in Σ', and hence $\sigma \in \Sigma'$. □

Thus, when checking that a data-independent program satisfies a data-independent property, it suffices to check that all differentiated traces of the program belong to the property. Hence, an observer for a data-independent property need only accept the differentiated traces that violate the property. This means that whenever a data value is input twice in a trace, the observer can stop checking (i.e., move to a non-accepting sink state), since the trace will anyway be ignored.

Note that the set of traces of a set is *not* data independent, e.g., since it contains a trace where two different data values are inserted, but *not* its renaming which inserts the same data value twice. This is not a problem, since the set of *all* traces of a set can be specified by observers, without using a data independence argument.

The key observation is now that the differentiated traces of queues and stacks can be completely and succinctly specified by observers with a small number of variables. We devote the following section to formalize and prove this fact.

5 Specifying stacks and queues using observers

We show in this section how to completely specify, using observers such as those introduced in Sect. 3, and using the data independence argument introduced in Sect. 4, the sequential behaviors of queues and stacks operating over the *arbitrary (and possibly infinite) data domain* $\mathbb{D}$. At the end of

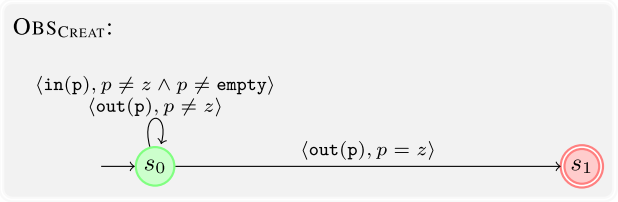


Fig. 6 A trace observer for checking that no data value can be extracted if it has not been inserted. The variable z is an observer variable, and `empty` is an observer constant

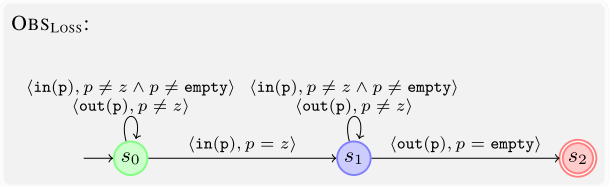


Fig. 7 A trace observer for checking that an inserted value has to be extracted before the data structure is declared empty. The variable z is an observer variable, and `empty` is an observer constant

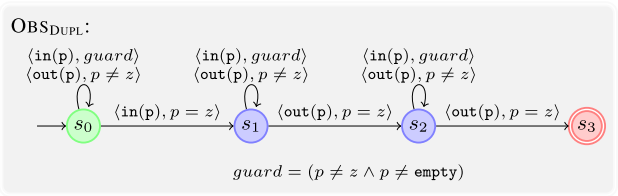


Fig. 8 A trace observer for checking that no once-inserted data value can be extracted twice. The variable z is an observer variable, and `empty` is an observer constant

this section, we will show that the three observers of Figs. 6, 7 and 8, in addition to the observer of Fig. 9 (respectively, Fig. 10) are enough to specify a stack (respectively, a queue) of arbitrary size[1]. We detail the approach for stacks and mention how to adapt it for the case of queues. First, we recall the natural operational specification of a stack and explain how we define its behavior using the set of traces it generates. Then, we propose, using the four simple observers mentioned above an alternative observational definition of a stack. The new definition abstracts away from the actual states and only considers properties of the generated traces. We write in the following $\underline{\mathbb{D}}$ to mean $\mathbb{D}\setminus\{\texttt{empty}\}$.

The functional specification of a sequential stack corresponds to the set of allowed finite sequences (we consider safety properties) of pushes and pops together with their arguments and return values. We use in the following $\texttt{in}(d)$ (respectively, $\texttt{out}(d)$ and $\texttt{out}(\texttt{empty})$) to mean a $\texttt{push}(d)$

[1] When the observers in Figs. 6, 7, 8, 10 and 9 are used to specify a stack (respectively, a queue), each occurrence of `in(.)` should be replaced by `push(.)` (respectively, `enq(.)`) and each occurrence of `out(.)` should be replaced by `pop(.)` (respectively, `deq(.)`)

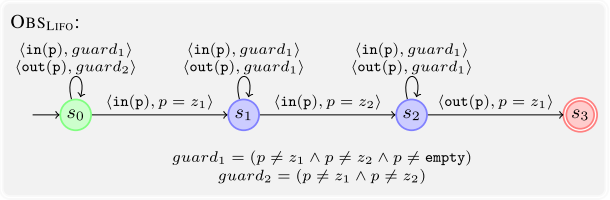


Fig. 9 An observer for detecting violations of the first inserted last extracted ordering. The initial state is s_0 and $\{s_3\}$ is the set of final states. The variables z_1, z_2 are observer variables, and `empty` is an observer constant

(respectively, $\texttt{pop}(\texttt{d})$ and $\texttt{pop}(\texttt{empty})$). The specification of a sequential stack is a strict subset of $(\Sigma_{i/o})^*$, where $\Sigma_{i/o} = \{\texttt{in}(d), \texttt{out}(d) \mid d \in \underline{\mathbb{D}}\} \cup \{\texttt{out}(\texttt{empty})\}$. We give in the following an operational and an observational characterization of the specification of a sequential stack and show their equivalence.

5.1 Operational specification of a stack

A natural way to define the set of finite stack traces is to use a transition system T where the set of states is the set of possible stack contents, and where the transitions are labeled with $\Sigma_{i/o}$. More formally, T is a tuple $(\Sigma_{i/o}, (\underline{\mathbb{D}})^*, \{\epsilon\}, \rightarrow)$, where the empty word $\epsilon \in (\underline{\mathbb{D}})^*$ is the initial state, and the set of transitions $\rightarrow \subseteq (\underline{\mathbb{D}})^* \times \Sigma_{i/o} \times (\underline{\mathbb{D}})^*$ only includes all transitions of the form: $\langle w, \texttt{in}(d), d\cdot w\rangle$, $\langle d\cdot w, \texttt{out}(d), w\rangle$, or $\langle \epsilon, \texttt{out}(\texttt{empty}), \epsilon\rangle$, where $d \in \underline{\mathbb{D}}$ and $w \in (\underline{\mathbb{D}})^*$. A run of T is a finite sequence $\rho = w_0 e_1 w_1 \cdots e_n w_n$ with $w_0 = \epsilon$ and $\langle w_i, e_{i+1}, w_{i+1}\rangle \in \rightarrow$ for each $i : 0 \leq i < n$. We say that ρ is a stack run. A trace of T is any sequence $e_1 \cdots e_n$ such that there is a stack run $w_0 e_1 w_1 \cdots e_n w_n$ of T. The operational specification of a stack, written ϕ^{op}_{stack}, is then the set of all traces of T.

Observe that the renaming of any stack trace is also a stack trace (just rename the states in the corresponding run). Also, given a trace σ resulting from a stack run ρ, one can obtain a differentiated trace whose renaming gives σ as follows. Repeat the same run, but append a systematically incremented counter to the values that are input to the stack. It is easy to see that the same run as ρ, except for the appended counter values to the data, is also a stack run on a differentiated trace that can be renamed (by forgetting the counter) into σ. The set of traces φ^{op}_{stack} therefore satisfies the definition of data independence introduced in Sect. 4.

As a result, Theorem 1 implies that any data-independent set of traces whose set of differentiated traces equals the set of differentiated stack traces does coincide with the set of stack traces. We write in the following $\varphi^{op}_{diff,stack}$ to mean the set of differentiated traces in φ^{op}_{stack}.

5.2 Observational specification of a stack

We propose another specification for differentiated stack traces, written $\varphi^{obs}_{diff,stack}$, which characterizes the set of differentiated stack traces as exactly those differentiated traces that are not accepted by any of four simple observers. Intuitively, such a differentiated trace satisfies the following four properties for all data values d_1 and d_2:

- NO CREATION ($\text{OBS}_{\text{CREAT}}$, Fig. 6): d_1 must not be popped before it is pushed, i.e., data cannot be created,
- NO LOSS (OBS_{LOSS}, Fig. 7)): $\texttt{empty}$ must not be returned if d_1 was pushed but not popped, i.e., data cannot be lost
- NO DUPLICATION (OBS_{DUPL}, Fig. 8): d_1 must not be popped twice, i.e., data cannot be duplicated.
- LIFO (OBS_{LIFO}, Fig. 9): d_2 must not be popped if d_1 was pushed after d_2 was pushed.

5.2.1 Differentiated operational and observational specifications coincide

Lemma 1 states that the differentiated operational specification of a stack equals the differentiated observational one.

Lemma 1 $\varphi^{op}_{diff,stack} = \varphi^{obs}_{diff,stack}$.

Proof Recall the claim only concerns differentiated traces. We will make use of two properties that hold for every stack run $\rho = w_0 e_1 w_1 e_2 \cdots e_n w_n$.

- The *counting property* of a stack. We write $(a)^{\#}_w$ to mean the number of occurrences of the letter $a \in A$ in the word $w \in A^*$, for a fixed alphabet A. Back to ρ, it is easy to show by induction that for every d in $\mathbb{D}$ and i s.t. $0 \leq i \leq n$, $(d)^{\#}_{w_i} = (\texttt{in}(d))^{\#}_{(e_1 \cdots e_i)} - (\texttt{out}(d))^{\#}_{(e_1 \cdots e_i)}$.
- The *ordering property* of a stack. Using the counting property and an induction on the length of ρ, one can show the following. Assume d_i and d_j are input before position k in ρ. If d_i is input before d_j, and if neither of them is output, then $w_k \in (\mathbb{D} \setminus \{d_i, d_j\})^* \cdot d_j \cdot (\mathbb{D} \setminus \{d_i, d_j\})^* \cdot d_i \cdot (\mathbb{D} \setminus \{d_i, d_j\})^*$.

We establish in the following inclusions in both directions in order to show the equality $\varphi^{op}_{diff,stack} = \varphi^{obs}_{diff,stack}$:

- $\varphi^{op}_{diff,stack} \subseteq \varphi^{obs}_{diff,stack}$. This direction is simple. Let $\rho = w_0 e_1 \cdots e_n w_n$ be a stack run giving a trace $\sigma = e_1 \ldots e_n$ in $\varphi^{op}_{diff,stack}$. Suppose σ is accepted by one of the observers OBS_{CREA}, OBS_{LOSS}, OBS_{DUPL}, or OBS_{LIFO} for some data values.
  1. σ cannot be accepted by OBS_{CREA}. Suppose it was the case and e_n is the $\texttt{out}(d)$ that labels the last transition in the observer. The fact that σ is accepted by the observer implies the data value d appearing in e_n does not participate in any $\texttt{in}(d)$ of the self-loop on s_0. The counting property implies $d \notin w_n$. Yet $e_n = \texttt{out}(d)$ requires w_n to be of the form $d \cdot w$.
  2. σ cannot be accepted by OBS_{LOSS} because the counting property implies w_{n-1} contains a d, yet $w_{n-1} = \epsilon$ since $e_n = \texttt{out}(empty)$ appears at the end of the stack run ρ .
  3. σ cannot be accepted by OBS_{DUPL} because the counting property requires w_{n-1} to contain no occurrences of d. Yet $e_n = \texttt{out}(d)$ requires w_{n-1} to be of the form $d \cdot w$.
  4. σ cannot be accepted by OBS_{LIFO} because that means ρ contains two events e_i, e_j with $1 \leq i < j < n$ such that $e_i = \texttt{in}(d_i)$ and $e_j = \texttt{in}(d_j)$. The ordering property of a stack implies that $w_{n-1} \in (\mathbb{D} \setminus \{d_i, d_j\})^* \cdot d_j \cdot (\mathbb{D} \setminus \{d_i, d_j\})^* \cdot d_i \cdot (\mathbb{D} \setminus \{d_i, d_j\})^*$. Yet for $e_n = \texttt{out}(d_i)$ to succeed, w_{n-1} needs to be of the form $d_i \cdot w$.
- $\varphi^{op}_{diff,stack} \supseteq \varphi^{obs}_{diff,stack}$. Suppose $\sigma = e_1 \ldots e_{n+1}$ in $\varphi^{obs}_{diff,stack}$ is a shortest trace not in $\varphi^{op}_{diff,stack}$. Hence, there is a stack run $\rho = w_0 e_1 \cdots e_n w_n$, but there is no w_{n+1} such that $\rho' = w_0 e_1 \cdots e_{n+1} w_{n+1}$ becomes a stack run.
  1. e_{n+1} cannot be $\texttt{in}(d)$ for some d because then it would be enough to choose $w_{n+1} = d \cdot w_n$ to get σ in $\varphi^{op}_{diff,stack}$.
  2. if $e_{n+1} = \texttt{out}(\texttt{empty})$, then $w_n \neq \epsilon$ as otherwise choose $w_{n+1} = \epsilon$ and σ would be in $\varphi^{op}_{diff,stack}$. Let $d \in w_n$. Using the counting property of a stack on ρ, we deduce that there is $e_i = \texttt{in}(d)$ for $i : 1 \leq i \leq n$, but $\forall j : 1 \leq j \leq n.\ e_j \neq \texttt{out}(d)$. Hence σ should have been accepted by OBS_{LOSS}, and therefore not in $\varphi^{obs}_{diff,stack}$,
  3. if $e_{n+1} = \texttt{out}(d)$ for some data value d:
     (a) If $\texttt{in}(d)$ does not appear in ρ, then σ should have been accepted by OBS_{CREA} and therefore it cannot belong to $\varphi^{obs}_{diff,stack}$
     (b) If $e_i = \texttt{in}(d)$ and $e_j = \texttt{out}(d)$ appear in ρ with $i, j : 1 \leq i, j \leq n$, then the counting property on the stack run ρ implies $i < j$. The trace σ should have been accepted by OBS_{DUPL}
     (c) If $e_i = \texttt{in}(d)$ appears in ρ for some $i \leq n$ but without a $e_j = \texttt{out}(d)$ for $i < j \leq n$, then the following holds. By the counting property, $w_n = w \cdot d \cdot w'$ with $w = d_k \cdot w''$. In addition, $d \neq d_k$ as otherwise ρ could be extended into a stack run. Using the counting property again, there must be a $e_k = \texttt{in}(d_k)$ with $k : 1 \leq k \leq n$ and without any $\texttt{out}(d_k)$ in the run up to n. If $i < k$, the trace should have been accepted by

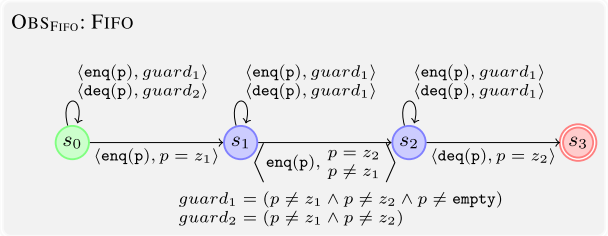


Fig. 10 An observer to check that FIFO ordering is respected. All unmatched abstract events, for example $\langle \texttt{deq(p)}, p = z_1 \rangle$, at location s_1, send the observer to a rejecting state

OBS_{LIFO}. If $k < i$, we use the ordering property to deduce that w_n should be in the language $(\underline{\mathbb{D}} \backslash \{d, d_k\})^* \cdot d \cdot (\underline{\mathbb{D}} \backslash \{d, d_k\})^* \cdot d_k \cdot (\underline{\mathbb{D}} \backslash \{d_1, d_2\})^*$ which contradicts that $w_n = d_k \cdot w'' \cdot d \cdot w'$. □

5.3 Operational and observational specification of queues

For a queue, $\texttt{in}(d)$ (respectively, $\texttt{out}(d)$ and $\texttt{out}(\texttt{empty})$) stands for $\texttt{enq}(\texttt{d})$ (respectively, $\texttt{deq}(\texttt{d})$ and $\texttt{deq}(\texttt{empty})$). The operational specification φ^{op}_{queue} is obtained by replacing $\rightarrow$ in Sect. 5.1 by the smallest subset of $\left((\underline{\mathbb{D}})^* \times \Sigma_{i/o} \times (\underline{\mathbb{D}})^*\right)$ that includes, for every $d \in \underline{\mathbb{D}}$ and $w \in (\underline{\mathbb{D}})^*$, all transitions of the form: $\langle w, \texttt{in}(d), w \cdot d \rangle$, $\langle d \cdot w, \texttt{out}(d), w \rangle$, and $\langle \epsilon, \texttt{out}(\texttt{empty}), \epsilon \rangle$. $\varphi^{op}_{diff,queue}$ is the restriction of φ^{op}_{queue} to the set of differentiated traces. The observational specification $\varphi^{obs}_{diff,queue}$ contains exactly those differentiated traces that are not accepted by any of the following four observers: OBS_{CREA}, OBS_{LOSS}, OBS_{DUPL}, or OBS_{FIFO} (Fig. 10). Intuitively, a differentiated trace that is not accepted by the observer OBS_{FIFO} satisfies the following property for any data values d_1, d_2:

FIFO (OBS_{FIFO}, Fig. 10): d_2 must not be dequeued if d_1 was not dequeued since it was enqueued before d_2 was enqueued.

Lemma 2 $\varphi^{op}_{diff,queue} = \varphi^{obs}_{diff,queue}$.

Proof Similar to the proof of lemma 1. We make use of the same counting property as in the stack case. We modify the ordering property to reflect the FIFO ordering (instead of the LIFO one for a stack). The other modifications are straightforward. □

6 Verification by shape analysis

To verify that no trace of the program is accepted by an observer, we form, as in the automata-theoretic approach [31], the cross-product of the program and the observer, synchronizing on abstract events, and check that this cross-product cannot reach a configuration where the observer is in an accepting state.

The analysis needs to deal with the challenges of an unbounded data domain, an unbounded number of concurrently executing threads, an unbounded heap, and an explicit memory management. As indicated in Sect. 1, the explicit memory management implies that the assertions generated by our analysis must be able to track correlations between pairs of threads.

For an illustration of the insufficient precision of pure thread-modular reasoning, consider a state of the program on Fig. 1 where one of the threads is about to execute line 4 of `enq`. It has allocated a new memory cell c pointed to by `n` and has set its `next` to `NULL`. Since thread-modular reasoning cannot relate states of local variables of different threads, it cannot express that actually no other thread with the control on line 4 of `enq` can have its variable `n` pointing at c. Suppose that there is another such thread. Let the first thread finish `enq`, going through lines 5–11 and 15, resulting in c being connected at the end of the queue with the global variable `T` pointing to it. The other thread, which still has its `n` at c, then goes through the **else** branch of the test on line 7 and starts another iteration of the while-loop. After passing through the tests on lines 7 and 8, the `CAS` on line 9 succeeds, redirecting the `next` pointer of c (which is the last cell of the queue) back to c. A configuration with a self-loop at the end of the queue is corrupt and may lead to all sorts of errors, all which would be false positives (consider, e.g., dequeuing the last element of such corrupt queue). Other false positives arises from that the pure thread-modular reasoning cannot express that two threads are never simultaneously about to free the same memory cell. This leads to reporting spurious double-free errors.

The reason for why presence of garbage collection matters is that with it, thread-modular analysis can be fixed not to report these false positives by means which cannot be used without it. Namely, to circumvent the problem in the first scenario, the analysis may keep within the state of a thread an information about cells that were freshly allocated and have not seen a global variable yet, such as the cell c. Due to garbage collection, such cells may be accessible by the allocating thread only. This information is sufficient to rule out scenarios such as the first one described above. This approach indeed works well in works such as [28]. The second type of situation involving double free does not arise simply because there are no frees under garbage collection. These fixes do not work without garbage collection. In the first situation, the exclusive access of the allocating thread to the cell c cannot be guaranteed because c could be freed earlier and now reallocated, with variables of other threads still pointing to it. Double free in the second situation is of course still an issue too. Our remedy to this, keeping correlations of local states

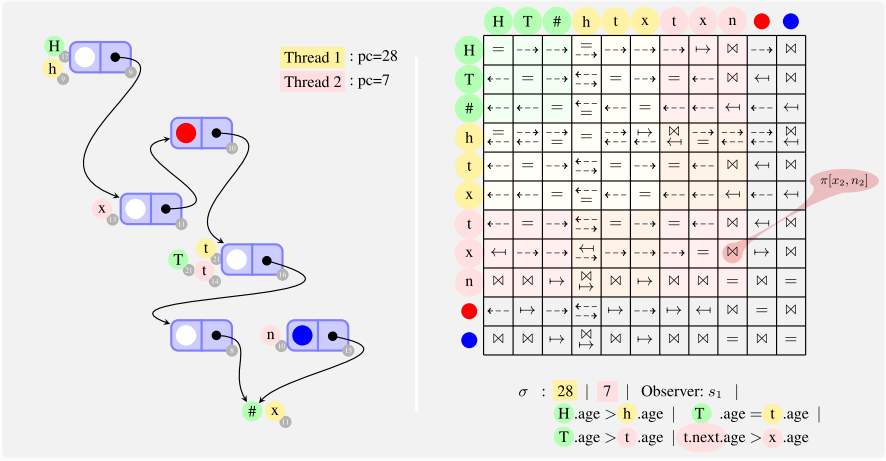


Fig. 11 Some concrete shape (to the *left*) and a saturated symbolic encoding (to the *right*) that denotes it (among many other concrete shapes). In this example, H and T denote some global variables, # denotes the NULL constant, the *small letters* correspond to local variables, and the *red* and *blue dots* refer to cells carrying data values to be tracked (those equal to the arbitrary values of some observer variables)

of pairs of threads, allows to express properties such as that certain memory cell is accessible by one thread only. It is enough to avoid both kinds of discussed false positives.

We now present our shape analysis in two steps. We first describe a symbolic encoding of the configurations of the program and then present the verification procedure.

6.1 Symbolic encoding

The symbolic encoding is used for characterizing the set of reachable configurations of the program from the point of view of two distinct executing threads. Roughly, this is done by recording the relationships of the local configurations of the two threads with each other, the relationships of the local variables of a thread with global variables, the observer configuration, and assertions about the heap. It is a combination of several layers of conjunctions and disjunctions.

Below, we will use Fig. 11 to explain the main concepts in the symbolic encoding. The left part of the figure shows a typical configuration of the heap that arises during an execution of the Michael & Scott algorithm, when run against the observer of Fig. 10. The right part of the figure shows a symbolic encoding that is satisfied by the shape. Note that the symbolic encoding can represent arbitrary more shapes. The concrete heap to the left consists of six cells operated on by two active threads, Thread 1 (depicted in yellow) and Thread 2 (depicted in pink). The threads are in control states 28 and 7, respectively, and the observer is in control state s_1. The topmost cell is pointed to by the global variable H and the local variable h of Thread 1. Each cell has a *data value* field and a *next* field, the latter being a pointer to the next cell in the heap. In our example, there are three possible values that can be stored in a cell, namely *red* which means that the value is equal to the value of variable z_1 of the observer, *blue* which means that the value is equal to the value of variable z_2 of the observer, and *white* which means that it is an arbitrary value different from the above two. The topmost cell has a data value which is *white*. Finally, the figure shows the counter values (i.e. *ages*) of all the pointers (those of the pointer variables and those of the *next* fields of the cells). For instance, the *next* pointer of the topmost cell has counter value 9, and the global variable H has counter value 17.

The right part of Fig. 11 depicts a symbolic encoding that is satisfied by the given configurations. More precisely, our symbolic encoding consists of two parts, the first part, called a *joined shape constraint*, given in matrix form, describes the *shape* of the heap, while the second part, called *control formula*, denoted by σ, gives the control states of the observer and the active threads, together with the relations that hold between the pointer counters. We now introduce the needed concepts one by one, in a bottom-up manner. Let us fix two thread identifiers i_1 and i_2.

Cell terms Let a *cell term* be one of the following: (i) a global pointer variable y, which denotes the cell pointed to by the global variable y, (ii) a term of the form $x[i_j]$ (where $j = 1$ or $j = 2$) for a local pointer variable x of thread i_j, which denotes the cell pointed to by the thread-i_j-local-copy of x, (iii) a special term NULL, UNDEF, or FREE, or (iv) a cell variable, which denotes a cell whose data value is equal to the current value of an observer variable. (Note that the value of an observer variable is fixed during a run of the observer). The latter allows us to keep track of the data in the heap cells, even in the case where a heap cell is not denoted by any pointer variable (in order to verify, e.g., the FIFO property of a queue). We use $CT(i_1, i_2)$ to denote the set of all cell terms (of thread i_1 and i_2).

Each row or column of the matrix in Fig. 11 is labeled by a cell term, e.g., T, n, #, etc. In particular, we use the red and blue circles, to denote the variables z_1 resp. z_2 of the observer.

Atomic heap constraint In order to obtain an efficient and practical analysis, which does not lead to a severe explo-

sion of formulas, we have developed a novel representation, adapted from the transitive closure logic of [6]. The representation is motivated by the observation that relationships between pairs of pointer variables are typically independent. The key aspect of the representation is that it is sufficient to consider only pairs of variables rather than correlating all variables. An atomic heap constraint is of one of the following forms (where t_1 and t_2 are two cell terms):

- $t_1 = t_2$: the cell terms t_1 and t_2 denote the same cell,
- $t_1 \mapsto t_2$: the `next` field of the cell denoted by t_1 denotes the cell denoted by t_2,
- $t_1 \dashrightarrow t_2$: the cell denoted by t_2 can be reached by following a chain of two or more `next` fields from the cell denoted by t_1,
- $t_1 \bowtie t_2$: none of $t_1 = t_2$, $t_1 \mapsto t_2$, $t_2 \mapsto t_1$, $t_1 \dashrightarrow t_2$, or $t_2 \dashrightarrow t_1$ is true.

We use $Pred$ to denote the set $\{=, \mapsto, \leftarrow\!\!|, \dashrightarrow, \dashleftarrow, \bowtie\}$ of all shape relational symbols. We let $t = \texttt{NULL}$ denote that t is null, $t \mapsto \texttt{UNDEF}$ denote that t is undefined, and $t \mapsto \texttt{FREE}$ denote that t is unallocated.

Each cell in the matrix of Fig. 11 contains a cell term. For instance, the cell pointed to by variable x of Thread 1 reaches in two or more steps the cell pointed to by variable t of Thread 2.

Joined shape constraint A joined shape constraint for two threads i_1 and i_2, denoted as $M(i_1, i_2)$, is a (typically large) conjunction $\bigwedge_{t_1,t_2 \in CT(i_1,i_2)} \pi[t_1, t_2]$ where $\pi[t_1, t_2]$ is a non-empty disjunction of atomic heap constraints. Intuitively, it is a matrix representing the heap parts accessible by the two threads (along with the cell data). Such a representation can be (exponentially) more concise than using a large disjunction of conjunctions of atomic heap constraints, at the cost of some loss of precision. In Fig. 11, the cell defined by the global variable T and the local variable h of Thread 1, indicates that *either* the cell pointed to by T is reachable from the cell pointed to by variable h of Thread 1, *or* the other way round. We say that a joined shape constraint $M(i_1, i_2)$ is *saturated* if for any three variables x, y, and z, the disjunction $\pi[x, z]$ does not contradict $\pi[x, x] \wedge \pi[x, y] \wedge \pi[y, y] \wedge \pi[y, z] \wedge \pi[z, z]$. Any joined shape constraint can be saturated by a straightforward fixpoint procedure, analogous to [6] or the one for DBMs [10]. For instance, let $\pi[x, y]$ be $x \mapsto y$, let $\pi[y, z]$ be $y \leftarrow\!\!| \; z \vee y \dashleftarrow z$, and let $\pi[x, x]$, $\pi[y, y]$, and $\pi[z, z]$ admit only equality (there is no loop involving x, y, or z). Then $\pi[x, z]$ can contain the disjuncts $x = z$ and $x \bowtie z$ because they are consistent with $x \mapsto y \wedge y \leftarrow\!\!| \; z$. It can also contain $x \mapsto z$, $x \dashrightarrow z$, and $x \bowtie z$ because they are consistent with $x \mapsto z \wedge y \dashleftarrow z$. The remaining predicates $x \dashleftarrow z$ and $x \leftarrow\!\!| \; z$, which are not consistent with $\pi[x, x] \wedge \pi[x, y] \wedge \pi[y, y] \wedge \pi[y, z] \wedge \pi[z, z]$, would be removed from $\pi[x, y]$ by saturation.

Symbolic encoding We can now define formally a symbolic encoding over two threads. A symbolic encoding is a disjunction $\Theta[i_1, i_2]$ of formulas of the form $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ where $\sigma[i_1, i_2]$ is a *control formula* and $\phi[i_1, i_2]$ is a *shape formula*.

A *control formula* $\sigma[i_1, i_2]$ contains (i) the current control location of threads i_1 and i_2, and the observer, and (ii) a conjunction encompassing the relations between the `age` fields of any pair of terms. For instance, when analyzing the program in Fig. 1, this conjunction includes among others, for a thread i, both relations `head[i].age`$\simeq$`Head.age` and `tail[i].ptr`$\rightarrow$`next.age`$\simeq$`next[i].age`, for $\simeq \in \{<, =, >\}$.

A *shape formula* $\phi[i_1, i_2]$ is a joined shape constraint conjoined with a formula $\psi[v_1, \ldots, v_m, z_1, \ldots, z_n]$ which links cell variables $v_1, \ldots, v_m$ with observer variables $z_1, \ldots, z_n$ that are used to keep track of heap cells with values equal to the observer variables. Formally, $\phi[i_1, i_2]$ is a formula of the form

$$\exists v_1, \ldots, v_m . [\psi[v_1, \ldots, v_m, z_1, \ldots, z_n] \wedge M(i_1, i_2)]$$

6.2 Verification procedure

We compute a program invariant of the form $\forall i_1, i_2 . (i_1 \neq i_2 \Rightarrow \Theta[i_1, i_2])$ which characterizes the configurations of the program from the point of view of two distinct executing threads i_1 and i_2. We obtain the invariant by a standard fixpoint procedure, starting from a formula that characterizes the set of initial configurations of the program. For two distinct threads i_1 and i_2, and for each control formula $\sigma[i_1, i_2]$, our analysis will generate one shape formula $\phi[i_1, i_2]$.

The fixpoint analysis performs a postcondition computation that results in a set of possible successor combinations of control and shape formulas. The new shape formulas of which the control formula already appears in the original $\Theta[i_1, i_2]$ will be used to weaken the corresponding old shape formula. Otherwise, if the control state is new, a new disjunct is added to $\Theta[i_1, i_2]$.

For two threads i_1 and i_2, we must consider two scenarios: either i_1 or i_2 performs a step, or some other (interfering) thread i_3, (distinct from i_1 and i_2), performs a step.

Postcondition computation In the first scenario, where one of the threads i_1 or i_2 performs a step, we can compute the postcondition of $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ as follows. $\sigma[i_1, i_2]$ is first updated to a new control state $\sigma'[i_1, i_2]$ in the standard way (by updating the possible values of control locations and observer state). $\phi[i_1, i_2]$ is then updated to $\phi'[i_1, i_2]$ by updating each conjunct $\pi[t_1, t_2]$ according to the particular program statement that the thread is performing. In general, we (i) remove all disjuncts that must be falsified by the step

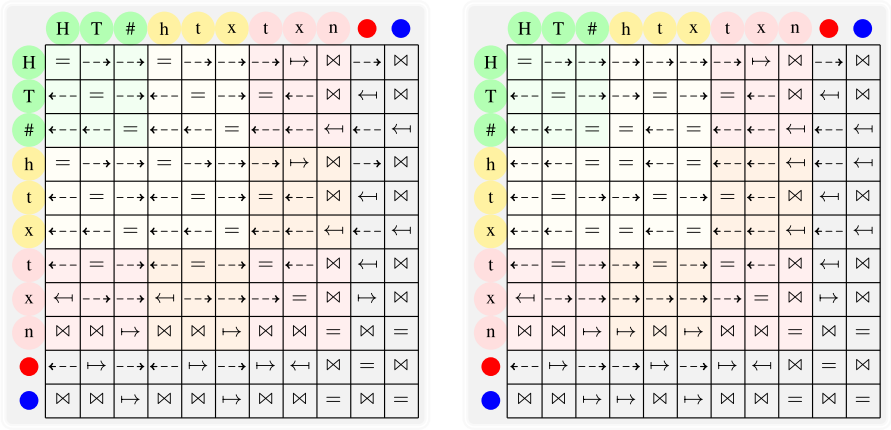


Fig. 12 Disambiguating the shape formula of the symbolic encoding in Fig. 11 when computing the image of `x.next:=n` by thread 2. Observe that this results in only two shape formulas, the left one where h_1 coincides with H and the right one where it equals `NULL`

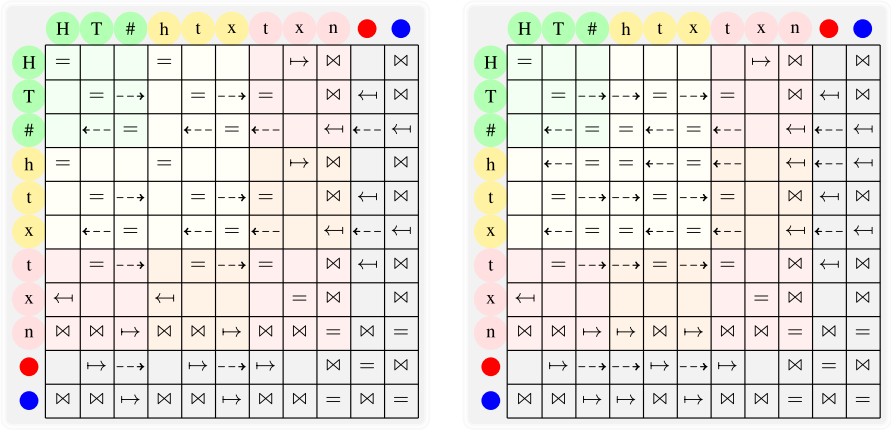


Fig. 13 Left and right shape formulas, respectively, obtained from the left and right formulas of Fig. 12 by removing all predicates that depended on the successor of the variable x_2 belonging to thread 2

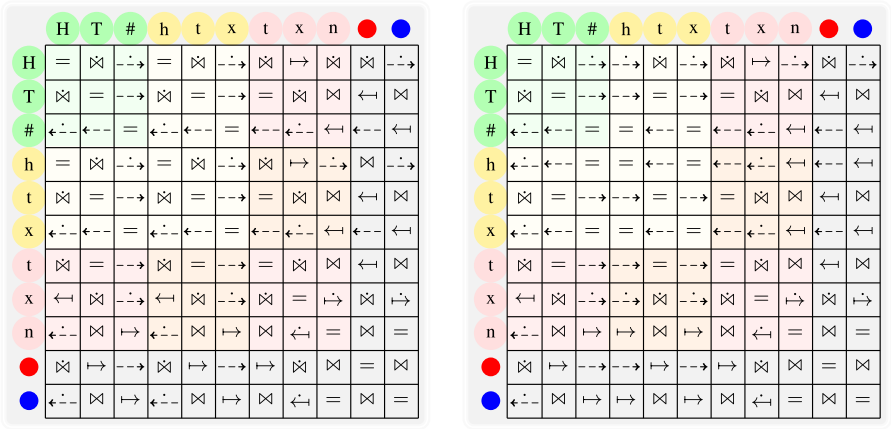


Fig. 14 Left and right shape formulas, respectively, obtained from the left and right formulas of Fig. 13 by updating with the predicate $x_2 \mapsto n_2$ together with all the resulting predicates. The obtained predicates are identified with a *dot* for presentation purposes. The resulting shape formula is then obtained by considering the cartesian abstraction of the two shape formulas

(this may require splitting the formula into several stronger formulas whenever the falsification might be ambiguous), (ii) add all disjuncts that may become true by the step, (iii) saturate the result.

Consider for instance the program statement `x:=y.next`. Since only the value of x is changing, the transformer updates only conjuncts $\pi[t, x]$ and $\pi[x, t]$ where $t \in CT(i_1, i_2)$. All assertions about x are reset by setting every conjunct $\pi[x, t]$ and $\pi[t, x]$ to $Pred$, for all $t \in CT(i_1, i_2)$. (The disjunction over all elements of $Pred$ is the assertion $true$). We then set $\pi[x, y]$ to $x \leftarrow\!\!\shortmid y$, $\pi[y, x]$ to $y \mapsto x$ and derive all predicates that may follow by transitivity. Finally, we saturate the formula. It prunes the (newly added) predicates that are inconsistent with the rest of the shape formula.

For `x.next:=y`, it is important to know the reachabilities that depend on the pointer x.`next`. The representation might potentially contain imprecision (it might for instance state that, for a term t, $\pi[t, x]$ contains $t \leftarrow\!\!- x$ and $t \dashrightarrow x$, even if we know, via a simpler analysis, that no cycles are generated). Hence, we first split the formula into stronger

formulas in such a way that we disambiguate the part of the reachability relation involving x (see Fig. 12). On each resulting formula, we then remove reachability predicates between cell terms that depend on $x.\texttt{next}$ (e.g., in Fig. 13 we remove $H \dashrightarrow T$ because $H \dashrightarrow x_2$ and $x_2 \dashrightarrow T$). We then set $\pi[x, y]$ to $x \mapsto y$ and derive all predicates that may follow by transitivity (e.g., in Fig. 14, since $x_2 \dashrightarrow n_2$ and $n_2 \dashrightarrow x_1$, we add $x_2 \dashrightarrow x_1$), and we saturate the result.

Interference In the case where we need to account for possible interference on the formula $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ by another thread, (distinct from i_1 or i_2), we proceed as follows. We (i) extend the formula with the interfering thread, (ii) compute a postcondition as described in the first scenario and (iii) project away the interfering thread.

Step (i) combines a given formula $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ with the information of an extra thread i_3. Like in [2], the resulting formula is of the form $(\sigma[i_1, i_2, i_3] \wedge \phi[i_1, i_2, i_3])$ such that any projection to two threads is a formula compatible with some disjunct of $\Theta[i_1, i_2]$. Intuitively, we consider an arbitrary third thread that is related to i_1 and i_2 using two of the binary relations generated so far; one relating i_1 to i_3 and another relating i_2 to i_3. In order to generate all such formulas involving three threads, we must, besides $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ itself, consider all pairs of disjuncts $(\sigma_\bullet[i_2, i_3] \wedge \phi_\bullet[i_2, i_3])$ and $(\sigma_\circ[i_1, i_3] \wedge \phi_\circ[i_1, i_3])$, such that $\sigma[i_1, i_2] \wedge \sigma_\bullet[i_2, i_3] \wedge \sigma_\circ[i_1, i_3]$ is consistent. Like $(\sigma[i_2, i_3] \wedge \phi[i_2, i_3])$, each of $(\sigma_\bullet[i_2, i_3] \wedge \phi_\bullet[i_2, i_3])$ and $(\sigma_\circ[i_1, i_3] \wedge \phi_\circ[i_1, i_3])$ is indeed some formula already generated by some sequence of post-computations or interference steps. In this case, we generate the formula $\sigma[i_1, i_2, i_3] \wedge \phi[i_1, i_2, i_3]$ where $\sigma[i_1, i_2, i_3] \equiv \sigma[i_1, i_2] \wedge \sigma_\bullet[i_2, i_3] \wedge \sigma_\circ[i_1, i_3]$ and $\phi[i_1, i_2, i_3] \equiv \phi[i_1, i_2] \wedge \phi_\bullet[i_2, i_3] \wedge \phi_\circ[i_1, i_3]$. We then saturate $\phi[i_1, i_2, i_3]$ (in the same way as for joined shape formulas over two threads). For each statement S of thread i_3 that can be executed when $\sigma[i_1, i_2, i_3]$ holds, we compute its postcondition $\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$ in step ii. Finally, $\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$ is projected back onto $\sigma'[i_1, i_2] \wedge \phi'[i_1, i_2]$ in step iii by removing all information about the variables of thread i_3.

Since the domain of control formulas and the domain of shape formulas over a fixed number of cell terms are finite, the abstract domain of formulas $\forall i_1, i_2.\ (i_1 \neq i_2 \Rightarrow \Theta[i_1, i_2])$ is finite as well. The iteration of postcondition computation is thus guaranteed to terminate.

7 Experimental results

We have implemented a prototype in OCaml and used it to automatically establish the conformance of concurrent data structures (including lock-free and lock-based stacks, queues and priority queues) to their operational specification (implying their linearizability). Our analyser also checks for standard pointer-related errors such as null or undefined pointer dereferencing (taking into account the known dangling pointers' dereferences [22]), double-free, or presence of cycles.

Some of the example programs are verified in the absence of garbage collection, in particular, the lock-free versions of Treiber's [27] stack and Michael & Scott's queue (see Fig. 1). We hereafter refer to them as Treiber's stack and M&S's queue, and garbage collection as GC. The verification of these examples is extensively demanding as it requires to correlate the possible states of the threads with high precision. We are not aware of any other method capable of verifying high level functionality of these benchmarks.

In addition to establishing correctness of the original versions of the benchmark programs, we also stressed our tool with few examples in which we intentionally inserted bugs (cf. Table 2). As expected, the tool did not establish correctness of these erroneous programs since the approach is sound. For example, we tested whether stacks (resp. queues) implementations can exhibit fifo (resp. lifo) traces, we tested whether values can be lost (loss observer), or memory errors can be triggered (memo observer accepts on memory errors made visible), we moved linearization points to wrong positions, and we tested a program which stores wrong values of inserted data. In all these cases, the analysis correctly reported traces that violated the concerned safety property. Finally, we ran the data structure implementations without garbage collection discarding the `age` counters and our (precise) analysis produced as expected a trace involving the ABA problem [18].

We ran the experiments on a 3.5 GHz processor with 8 GB memory. We report, in Table 1, the running times (in seconds) and the final number of joined shape constraints generated ($|C|$, reduced by symmetry).

We also include a succinct comparison with related work. Although it is often unfair to compare approaches solely based on running times of different tools, we believe that such a comparison can give an idea of the efficiency of the involved approaches. Our running times on the versions of Treiber's stack and M&S's queue that assume GC are comparable with the results of [30]. However, the verification of versions that do not assume GC is, to the best of our knowledge, beyond the reach of [30] (since it does not correlate states of different threads). The work in [25] verifies linearizability of concurrent implementations of sets, e.g., a lock-free `CAS`-based set [32] (verified in 2975s) of a comparable complexity to M&S's queue without GC (550s with our prototype). Basic memory safety of M&S's queue and two-locks queue [21] without GC was also verified in [36], but only for a scenario where all threads are either dequeuing or enqueuing. The verification took 727 and 309 s for M&S's queue and 6162 and 304 s for the two-locks queue. Our verification analysis produced the same result significantly faster, even allowing every thread to

Table 1 Experimental results

Data structure	Conformance			Safety only	
	Observers	Time (s)	$\|C\|$	Time (s)	$\|C\|$
Coarse Stack	Stack+	0.02	436	0.01	102
Coarse Stack, no GC		0.07	569	0.01	130
Coarse Queue	Queue+	0.04	673	0.01	196
Coarse Queue, no GC		0.48	1819	0.10	440
Two-Locks Queue [21]	Queue+	0.08	1830	0.02	488
Two-Locks Queue, no GC		0.73	3460	0.13	784
	vs 47 s in [5]			vs 6162 s/304 s in [36]	
Coarse Priority Queue (Buckets)	Prio	0.24	1242	0.07	526
Coarse Priority Queue (List-based)		0.04	499	0.01	211
Bucket locks Priority Queue		0.22	1116	0.05	372
Treiber's lock-free stack [27]	Stack+	0.23	714	0.01	78
	vs 0.09 s in [30]				
Treiber's lock-free stack, no GC	Stack+	2.28	1535	0.10	190
	vs 53 s in [5]				
M&S's lock-free queue [21]	Queue+	3.31	3476	0.44	594
	vs 3.36 s in [30]				
M&S's lock-free queue, no GC	Queue+	550	53,320	25	6410
	vs o.o.m. in [5]			vs 727 s/309 s in [36]	

Stack+ (resp. queue+) is an observer encompassing the loss, creation, duplication and lifo (resp. fifo) observers

Table 2 Introducing intentional bugs: the analysis is sound and the programs are not verified

Data structure	Modification	Observer	Output	Time (s)
Treiber's stack	None	Fifo	Bad trace	0.07
Treiber's stack, no GC	None	Fifo	Bad trace	6.19
M&S's queue	None	Lifo	Bad trace	1.26
Two-locks queue	Bad commit point	Fifo	Bad trace	0.02
M&S's queue	Bad commit point	Loss	Bad trace	0.51
Treiber's stack	Omitting data	Lifo	Bad trace	0.02
Treiber's stack, no GC	Discard ages	Loss	Bad trace	0.42
Treiber's stack, no GC	Discard ages	Loss	Cycle creation	0.01
M&S's queue, no GC	Discard ages	Loss	Bad trace	272
M&S's queue, no GC	Discard ages	Loss	Dereferencing null	0.01
M&S's queue	Swapped assignments	Memo	Dereferencing null	0.01

non-deterministically decide to either enqueue or dequeue. In [5], linearizability of the Treibers's stack (resp. two-locks queue [21]) is verified in 53 s (resp. 47 s). We achieve the same result in less than 3 s. Finally, a variant of M&S's queue without GC could not be successfully verified in [5] due to lack of memory.

8 Conclusions and future work

We have presented a technique for automated verification of temporal properties of concurrent programs, which can handle the challenges of infinite-state specifications, an unbounded number of threads, and an unbounded heap managed by explicit memory allocation. We showed how such a technique can be based naturally on the automata-theoretic approach to verification, by nontrivial combinations and extensions that handle unbounded data domains, unbounded number of threads, and heaps of arbitrary size. The result is a simple and direct method for verifying correctness of concurrent programs. The power of our specification formalism is enhanced by showing how the data-independence argument by Wolper [35] can be introduced into standard program analysis. Our method can be parameterized by different

shape analyses. Although we concentrate on heaps with single selectors in the current paper, we expect that our method can be adapted to deal with multiple selectors, by integrating recent approaches such as [16]. Moreover, our experimentation deals with the specification of stacks and queues. Other data structures, such as deques, can be handled in an analogous way.

References

1. Abdulla, P., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for LTL(MSO). STTT **14**(2), 223–241 (2012)
2. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few. In: VMCAI, pp. 476–495. Springer, Berlin (2013)
3. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: TACAS, vol. 7795, LNCS, pp. 324–338. Springer, Berlin (2013)
4. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Proc. of CAV'07. LNCS, vol. 4590, pp. 477–490. Springer, Berlin (2007)
5. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S.: Thread quantification for concurrent shape analysis. In: Proceedings of CAV'08. LNCS, vol. 5123, pp. 399–413. Springer, Berlin (2008)
6. Bingham, J., Rakamaric, Z.: A logic and decision procedure for predicate abstraction of heap-manipulating programs. In: Proc. of VMCAI'06. LNCS, vol. 3855, pp. 207–221. Springer, Berlin (2006)
7. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Proceedings of PLDI'10, pp. 330–340. ACM, New York (2010)
8. Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Proc. of CAV'10, LNCS, vol. 6174, pp. 465–479. Springer, Berlin (2010)
9. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: Proceedings of CAV'06. LNCS, vol. 4144, pp. 475–488. Springer, Berlin (2006)
10. Dill, D.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite-State Systems, vol. 407. LNCS. Springer, Berlin (1989)
11. Doherty, S., Detlefs, D., Groves, L., Flood, C., Luchangco, V., Martin, P., Moir, M., Shavit, N., Jr. G.S.: Dcas is not a silver bullet for nonblocking algorithm design. In: Proceedings of SPAA'04, pp. 216–224. ACM, New York (2004)
12. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: Proceedings of FORTE'04. LNCS, vol. 3235, pp. 97–114. Springer, Berlin (2004)
13. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. In: Proceedings of TACAS'10, vol. 6015. LNCS, pp. 296–311. Springer, Berlin (2010)
14. Emmi, M., Jhala, R., Kohler, E., Majumdar, R.: Verifying reference counting implementations. In: Proceedings of TACAS'09. LNCS, vol. 5505, pp. 352–367. Springer, Berlin (2009)
15. Flanagan, C., Freund, S.: Atomizer: a dynamic atomicity checker for multithreaded programs. Sci. Comput. Program. **71**(2), 89–109 (2008)
16. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: Formal Methods in System Design, pp. 1–24 (2012)
17. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
18. IBM. System/370 principles of operation (1983)
19. Kidd, N., Reps, T., Dolby, J., Vaziri, M.: Finding concurrency-related bugs using random isolation. STTT **13**(6), 495–518 (2011)
20. Michael, M., Scott, M.: Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Rochester (1995)
21. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of 15th ACM Symposium on Principles of Distributed Computing, pp. 267–275 (1996)
22. Michael, M.M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, PODC '02, pp. 21–30. ACM, New York (2002)
23. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proceedings of PLDI'06, pp. 308–319. ACM, New York (2006)
24. Naik, M., Park, C.-S., Sen, K., Gay, D.: Effective static deadlock detection. In: Proceedings of ICSE, pp. 386–396. IEEE, New York (2009)
25. Segalov, M., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Abstract transformers for thread correlation analysis. In: APLAS, LNCS, pp. 30–46. Springer, Berlin (2009)
26. Shacham, O.: Verifying atomicity of composed concurrent operations. PhD thesis, Department of Computer Science, Tel Aviv University (2012)
27. Treiber, R.: Systems programming: coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr. (1986)
28. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Proceedings of VMCAI, vol. 5403. LNCS, pp. 335–348. Springer, Berlin (2009)
29. Vafeiadis, V.: Automatically proving linearizability. In: CAV, vol. 6174. Lecture Notes in Computer Science, pp. 450–464. Springer, Berlin (2010)
30. Vafeiadis, V.: Rgsep action inference. In: Proceedings of VMCAI'10, vol. 5944. LNCS, pp. 345–361. Springer, Berlin (2010)
31. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of LICS'86, pp. 332–344 (1986)
32. Vechev, M., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Proceedings of PLDI'08, pp. 125–135. ACM, New York (2008)
33. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proceedings of SPIN'09, vol. 5578. LNCS, pp. 261–278. Springer, Berlin (2009)
34. Wang, L., Stoller, S.: Static analysis of atomicity for programs with non-blocking synchronization. In: Proceedings of PPOPP'05, pp. 61–71. ACM, New York (2005)
35. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic (extended abstract). In: Proceedings of POPL'86, pp. 184–193 (1986)
36. Yahav, E., Sagiv, S.: Automatically verifying concurrent queue algorithms. Electr. Notes Theor. Comput. Sci. **89**(3) (2003)