Symbolic Representations of Dynamic Data Structures in Program Analysis

Habilitation Thesis

Adam Rogalewicz

Brno 2016

Abstract

In this habilitation thesis, we discuss the problem of verification of programs with pointers and dynamic memory allocation. One of the main components needed in verification of these programs is a suitable representation of a (potentially infinite) set of memory configurations often called shape graphs. Shape graphs are in fact unrestricted oriented graphs, and, moreover, we have to work with infinite sets of these objects. A suitable symbolic model of memory configurations must satisfy the following properties: First, it must describe in a finite way an infinite number of shape graphs. Second, it must allow an execution of program statements on the level of the symbolic model. And finally, some acceleration technique must exists to guarantee a termination of the execution on the level of the used symbolic model. In this thesis, we discuss various types of automata and logics as natural tools to represent infinite sets of objects, which can be used as symbolic models of shape graphs. In the area of automata, we describe use of finite automata, tree automata, forest automata, counter automata, and we discuss also the concept of graph automata. In the area of logics, we concentrate mostly on monadic second-order logic and separation logic. The main part of this thesis is a collection of original research papers, where the author of this thesis is the key contributor.

Keywords

Formal verification, shape analysis, finite automata, tree automata, separation logic, monadic second-order logic, decidability.

Acknowledgment

I would like to thank to Prof. Tomáš Vojnar, the head of our research group for excellent research cooperation, valuable suggestions, and financial support. Then I would like to thank to all my colleagues and co-authors for excellent cooperation leading to all the results, in particular to Prof. Ahmed Bouajjani, Dr. Peter Habermehl, Dr. Lukáš Holík, Dr. Radu Iosif, Dr. Ondřej Lengál, and Dr. Jiří Šimáček. And finally, I would like to thank to my wife Romana for her great moral support and a lot of tolerance in the life with a scientist.

Over the time, the author was supported by number of grant projects. In particular by Czech Science Foundation under the projects 102/07/0322, 201/09/P531, 103/10/0306, 14-11384S and 16-17538S, the French project RNTL AVERILES, Czech-French projects BARRANDE MEB021023 and MEB020840, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), COST IC0901, and internal BUT projects FIT-S-11-1, FIT-S-12-1, and FIT-S-14-2486.

Contents

Chapter 1

Introduction

Computers and computer-controlled systems are nowadays integral part of everyday life. Almost everyone has a personal computer and other computer powered personal devices such as a mobile phone, a DVD player, or a smart TV set. Moreover, computers are used as real-time controllers embedded in cars, planes, train control systems, medical devices or power plants. During the years, people got used to rely on computers as error-free devices which do its work even better than humans. However, a problem is that computer systems are not error-free, they contain plenty of errors which may appear after a long time of reliable usage. An error in a critical system may cause a huge financial loss and sometimes even fatalities[1]. Contrary to humans, a computer often does not recognize that it made an error, and if so, it is incapable to improvise to bypass the consequences. At the same time, the size of computer systems is growing, and the systems themselves become more and more complex. Together with the rising size and complexity, the probability of a critical error is also rising. Therefore automated techniques to improve, and even guarantee, reliability of computer systems (including hardware and software) are strongly demanded. The frequently used techniques are various methods of intelligent testing and formal verification.

1.1 Testing

Testing is a widely used technique for improving reliability of computer systems. The naive way is to take a computer system and run it many times with different input data. Whenever an error is detected, the tester is sure that it is a real bug, and, moreover, it can be often easily reproduced[2]. But the naive testing methods stop to work when the number of problematic inputs (i.e. the inputs leading to an error behavior) is very low in comparison with the correctly working inputs. The problematic inputs may be related, e.g., to a single branch of a program which is almost never executed. Many more problems arise in concurrent programs, where the particular threads can be mutually scheduled in

[1] CBCNews: Toyota "Unintended Acceleration" Has Killed 89,
`http://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/`

[2] A problem is to reproduce bugs in parallel and other nondeterministic systems

many different ways and only some of these schedules can cause problematic behavior such as deadlocks, data races, or livelocks—see. e.g. [FKLV12].

Therefore various intelligent methods of testing were proposed [AO08]. Most of them are based on code coverage metrics, and the tester (or automated testing engine) is producing inputs such that he/she achieves (almost) full coverage of the tested system. Coverage metrics for single threaded programs as, e.g., path coverage or loop coverage, are quite well established. In the case of concurrent programs, the situation is much more complicated due to the number of possible mutual schedulings of parallel processes. Moreover, various concurrent coverage metrics were also established recently—see e.g. [Let12].

The latest advances in testing allow programmers to produce computer systems with a very low probability of error behavior. Nevertheless, despite of the huge progress in various testing methods, testing cannot (in most of the cases) guarantee correctness for all possible inputs. This is not such a big problem in production software designed for personal devices, but it is a huge problem for computer systems serving as real-time controllers of critical applications. Therefore methods for proving program correct w.r.t. some specification are welcome.

1.2 Formal Verification

By *formal verification*, we understand methods based on a formal mathematical basis designed to prove or disprove whether a system satisfies given requirements. In the case of disprovement, it is useful when the verification method provides a concrete counterexample run leading to an error behavior. Techniques of formal verification differ in the used mathematical principles, level of automation, constructions allowed in the checked system and types of properties which can be checked.

Computer systems can be roughly divided into ones with finite state space and ones with infinite state space. For a system with a finite state space, one can (at least in theory) enumerate all possible states reachable from the initial one and then check whether a given property holds. But when the size of the state space starts to be very large or even infinite, it is impossible to explicitly enumerate all the reachable states. At the same time, systems with infinite states space are quite natural in practice. The infinity can be caused, e.g., by a use of unbounded data structures (stacks, queues, etc.), real time, or unrestricted parallelism. To deal with infinite state spaces, one can use techniques of automated abstractions [GS97, BBLS00, BLBS01, McM06], cut-offs [GS92, EN96], automated induction [WL89, KM95, MQS00, LHR97, CR00, PRZ01] or symbolic verification [BCM$^+$92, KMM$^+$01, BJNT00]. The last possibility is the one tackled in our research and in this thesis.

Symbolic verification

In symbolic verification, a large or infinite number of states is represented in a compact symbolic formalism such as, e.g., binary decision diagrams [BCM$^+$92], various types of automata [BJNT00, Abd12, Leg12, BHRV12], logics [Rey02, SRW02, MPQ11, MS01],

or grammars [LYY05, HJKN15]. For each concrete step of a verified system, one has to provide a symbolic transformer manipulating the possibly infinite number of configurations on the level of the symbolic representation. In order to make the symbolic verification terminate in case of infinite state space systems, one has to employ some acceleration technique such as widening [BJNT00, BT12] or automated abstraction [BHV04, BHRV05, BHRV12] that overapproximates the set of reachable states.

1.3 Programs with Pointers and Dynamic Memory Allocation

Dynamically allocated data structures linked by pointers are a widely used construct in low-level programming languages such as the C language. They present a powerful technique which allows one to implement data structures with variable sizes, such as lists, stacks, or trees. Pointers are often used in system code, such as operating system kernels, standard libraries, or implementation of data containers, used in higher-level programming languages, such as C++ or Java. However, the use of pointers may be quite intricate, hence this kind of programs can easily contain hidden errors. The most common bugs in pointer programs are the following:

- *Null and undefined pointer dereference:* the program tries to access a memory cell using null or invalid pointer value.
- *Garbage:* there are parts of allocated memory inaccessible from the program variables.
- *Data structure inconsistency:* the data structure does not satisfy the considered property—e.g., in doubly-linked lists, $x \to next \to back$ is not equal to x.

Due to the intricate behavior of pointer programs and the usage of pointers in critical system code in general, techniques for automated analysis are quite needed here. The memory configuration of a pointer program is in fact an unrestricted oriented graph (often called a *shape-graph*), where nodes are allocated memory cells and edges represent pointer links between these cells. Hence, for automated analysis, one needs to be able to handle unbounded sets of unrestricted graphs.

Plenty of techniques for symbolic representation of shape-graphs were proposed by numerous research teams. Despite the progress in the last 20 years, the state-of-the-art is still far from satisfactory. The techniques can be divided into three main categories as follows:

- *Automata based:* a set of shape graphs is represented by some kind of (finite-state) machine, such as finite automata, tree automata, or forest automata—see Chapter 2.
- *Logic based:* a set of shape graphs is represented using a logical formula in some suitable logic, such as separation logic or 3-valued predicate logic with transitive closure—see Chapter 3.

- *Other approaches:* the shape graphs can also be represented and manipulated by graph grammars [LYY05, HJKN15], memory patterns [YKB02, ČEV07b, ČEV07a], Petri nets [BCK01], or other techniques as, e.g., [WKZ+06, CR07, LAIS06].

1.4 Structure of the Thesis

The rest of the thesis is organized as follows: Chapter 2 presents a brief introduction into automata-based techniques for representation and manipulation of shape graphs, Chapter 3 describes logic-based techniques, Chapter 4 summarizes contributions of the thesis, and, finally, Chapter 5 concludes the thesis and discusses the possible directions of the future research.

Chapter 2

Automata-based Techniques for Analysis of Pointer Programs

Various kinds of automata are commonly used in computer science as finite representations of infinite sets of objects. In order to represent sets of shape graphs (i.e. memory configurations of pointer programs)[1] it is a natural way to represent them by some kind of graph automata as, e.g., [MP11, Tho91]. The concept of graph automata used in [MP11] with their full definition is presented in Sec. 2.4. Unfortunately, due to the complexity and undecidability issues related to emptiness and inclusion problems, graph automata cannot be easily employed in verification. Therefore several encodings of graphs into words, trees, forests, or even vectors of integers were proposed in order to use finite word, tree or counter automata. This chapter presents a short overview of these encodings.

2.1 Finite (Word) Automata

In this section, we introduce two remarkable applications of finite automata to analysis of pointer programs.

One Selector Linked Lists in Regular Model Checking

In [BHMV05], the authors proposed an encoding of programs manipulating linked lists into the framework of regular model checking [BHV04]. The idea of the encoding is based on the fact that a garbage-free linked list can be always split into a finite number of uninterrupted singly-linked segments[2]. Moreover, the number of these segments is linear in the number of pointer variables. These segments are then linearized into a word, pointer variables are placed directly at the places where they point-to, and references between these segments are encoded by means of so-called *marker pairs*. A marker pair contains a *from_marker* m_f placed at the end of a segment and a *to_marker* m_t placed on

[1] As was already mentioned in Sec. 1.3, a memory configuration of a pointer manipulating program can be viewed as an unrestricted oriented graph with edges labeled by selectors.

[2] A segment may be interrupted by a node pointed by a pointer variables or by a node representing a join of two segments.

an arbitrary position except the end of a segment. A from_marker m_f represents the starting point of the edge leading to the corresponding to_marker m_t. There can only be a finite number of segment, which implies a finite number of marker pairs in the alphabet.

Example: Fig. 2.1 shows a memory configuration contains of three uninterrupted singly-linked segments. This configuration can be encoded as a word $x///m_f^1|ym_t^2///m_f^1|m_t^1//m_f^2|$ where / states for $next$ selector and | is a segment separator.

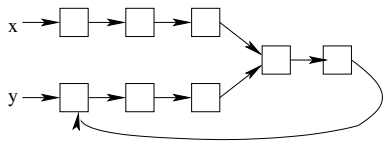


Figure 2.1: An example memory configuration.

□

A regular set of heaps can then be naturally encoded as a finite automaton, pointer updates are encoded as finite state transducers and the technique of regular model checking can be used to compute (an overapproximation of) the set of reachable configurations.

Automata in May-Alias Analysis

The work [Ven99] (continuation of earlier works [Jon81, Deu94]) considered the special problem of *may-alias analysis*—an analysis used for discovering a potential aliasing between pointers. In particular, this analysis searches for pointers and pointer sequences which can point to the same location during a program execution. Memory structures are represented as tuples of finite automata (one automaton for each pointer variable) and an alias relation. To accelerate the fixpoint computation, a special widening designed for these structures was proposed.

2.2 Tree Automata

In this section, we introduce three different approaches to encode dynamic data structures by tree automata. The first one based on *Tree Automata and Graph Types* [BHRV06] is the most general one, but the encoding can easily break down after higher number of destructive updates performed on an unbounded loop. The second one based on *Forest Automata* [HHR$^+$11] is less expressive then the first one, but the encoding is robust due to ideas of local reasoning inspired by Separation logic (see Sec. 3.2). The last encoding based on *Top-Down Parity Tree Automata* [DEG06] allow one to encode wide class of properties to be verified, but the programming language is restricted to a single procedure without nested loops.

Tree Automata and Graph Types

In [BHRV06], we proposed an original encoding based on tree automata combined with graph types [KS93]. In graph types, nodes are linked by two types of edges: *standard edges* and *extra edges*. The standard edges form a tree and the extra edges are described by regular expressions over an alphabet based on two components: (i) directions in the tree composed from the standard edges and (ii) types of nodes (node with k next selectors, leaf, root, etc.). These regular expressions are placed into nodes from where the extra edges originate.

In mode detail, the main idea of the above tree encoding of a graph is to use a *tree backbone* and a finite number of routing expressions. A node with n outgoing selectors is then encoded by an alphabet symbol from $2^{Var} \times 2^M \times (D \cup E \cup \{null, undefined\})^n$. Var is a set of pointer variables, D is a set of directions in the backbone tree, where a symbol $x \in D$ represents the fact that the edge leads to the x^{th} son of the node (0 is used for the parent). E is a finite set of references to routing expressions (stored externally of the tree encoding). Each routing expression is a regular expression over $\Sigma_R = D \cup \{leaf, root\}$ and it is coupled with a marker from M. Let $e_i \in E$ be a reference to a routing expression placed in a node p in the backbone tree. Then the destination of an e_i-labeled edge is a node r marked with a corresponding marker $m_i \in M$ such that the path from the node p to the node r satisfies the routing expression coupled with the reference e_i. The number of routing expressions is usually fixed to the number of destructive pointer updates ($x \to s = y$) in the program such that each destructive update has its own routing expression.

Example: As an example, let us take a binary tree with root pointers where pointer variable x points to the root. The data structure and its encoding are depicted in Fig 2.2. The reference E_1 corresponds to the routing expression 0^* and is coupled with the marker M_1.

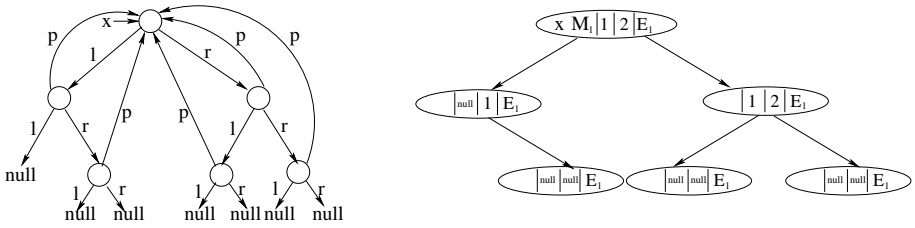


Figure 2.2: A binary tree with parent pointers (left) and its encoding based on backbone tree and the routing expression $E_1 = 0^*$. □

Above, we spoke about an encoding a single graph. When one need to encode a set of graphs, one have to capture a set of tree backbones and the routing expressions. Regular sets of tree backbones are symbolically captured by a TA and each routing expression by a TA as well. Program statements are then encoded as tree transducers with a single exception—a destructive update is encoded as a tree transducer coupled with routing expression update. This encoding allows one to use the technique of regular (tree)

model checking to compute an overapproximation of reachable states [BHRV12][3]. The technique was applied on a various kind of small programs manipulating non-trivial data structures such as DLLs, or various kinds of trees including trees, where leaves are linked into a singly-linked list. The main disadvantage of this approach is that the tree backbone is fixed during the whole computation. If destructive updates performed on a loop does not fit the idea of routing expressions then the routing expression will easily become Σ_R^* and false positives will appear as a consequence.

In [IR13][4], the overapproximation of reachable states was used to translate the original pointer program into a Büchi automaton allowing one to prove its termination by reduction to the language emptiness problem of Büchi automata.

Forest Automata

In [HHR$^+$11][5], we proposed a concept of Forest automata. The crucial idea is that each oriented graph with a finite number of input nodes (e.g. nodes pointed by pointer variables) can be decomposed into an ordered forest where the root of each forest is either (i) an input node or (ii) a node with more then one incoming edge. Both the nodes of type (i) and (ii) are called cutpoints. Each leaf of the tree may contain $null$ or a reference to a root of some component in the forest. The semantics of a forest is an oriented graph obtained by gluing all root references with the corresponding roots.

A forest automaton (FA) is then a tuple of tree automata accepting a graph language. A graph is a part of the language iff it can be decomposed into an ordered forest where the i^{th} tree is accepted by the i^{th} tree automaton from the FA.

A single FA accepts a graph language where all graphs have an equal number of cutpoints. Hence, sets of forest automata (SFA) are used instead of a single FA to cover graph languages with various numbers of cutpoints. The SFA are closed under union and intersection and their inclusion problem is decidable.

The SFA cover a wide class of graph languages but its disadvantage is that the number of cutpoints must be bounded. Hence, an SFA is not enough to represent, e.g., the language of all DLLs (doubly-linked lists) of an arbitrary size. Each node in a DLL (except for the head and tail nodes) is a cutpoint with two incoming edges. In order to cover graph languages with an unbounded number of cutpoints, hierarchical FA were introduced. In hierarchical FA (resp. SFA), the accepted graphs can contain special k-ary hyperedges called *boxes* with one node called the *input port* and k nodes called *output ports*. Such edges can be simply used in the rules of TA. Each box represents a reference to a graph language represented again by a hierarchical SFA where the input nodes correspond to the ports of the box. A strict hierarchy is required[6]. Hierarchical SFA are closed w.r.t. union, and a sound semi-algorithm for inclusion checking was proposed for them. They are used as a symbolic model in the shape analysis tool called Forester and the tool was

[3] Appendix C

[4] Appendix A

[5] Appendix B

[6] This restriction can be in theory lifted, but as a consequence, some pointer updates can not be symbolically executed any more.

applied on a set of small, but pointer intensive, programs manipulating various kinds of lists (including skip-lists) and trees. The boxes needed to verify a certain program (or class of programs) were provided manually in the first version of the tool, and in the later versions, a heuristics is used to learn them automatically [HLR+13][7].

Top-Down Parity Tree Automata

Another automata based approach was proposed in [DEG06]. It is based on parity tree automata (top-down tree automata with the parity acceptance condition). These automata work on memory shape graphs unfolded into infinite trees in a natural way. The principle allows one to encode wide range of properties like reachability of a memory node, cycles in the memory, sharing, dangling pointers, null pointer dereferences, etc. The main limitation is that the programming language is rather restricted—the technique can be applied on a single procedure without a call to other procedures, the verified procedure must have a special pointer called *cursor* and all modifications of the memory configuration are done in a finite neighborhood of it, and no nested loops are allowed. The procedure is automatically translated into the considered automata. Then, a product of the automaton describing the input data structure, the automaton describing the procedure, and the bad property automaton is computed. The resulting automaton is then checked for emptiness.

2.3 Simulation by Counter Automata

A counter automaton (CA) is a finite automaton equipped with a finite set of integer variables called counters. A transition between two states is labeled by a logical *expression* on current and next values of counters, where the next value of a variable x is usually denoted as x'. The expression encodes both a guard and an action. A counter automaton can perform any move satisfying the particular expression on a transition. A run starts in an initial state with unconstrained values of variables and terminates when no transition can be fired. A CA is a natural model for integer programs. However, as shown below, it can be used to encode and analyze programs working with other data too.

The reachability and termination problems for CA are in general undecidable, but various successful semi-algorithmic approaches were proposed as, e.g., ARMC [Ryb], Terminator [CPR06], or FLATA [BIKV].

Let us now have a look how to use CA for dealing with pointer programs. The main advantage is that one can verify properties such as, e.g., termination of the procedure, or balanceness of a tree. On the other hand one have to analyze machines with unbounded counters with all disadvantages related to it. The basic idea of using CA for verification of programs with pointers is to establish numerical measures (sometimes called norms) on data structures such as the length of the list segment pointed by a variable x, the number of nodes in a tree, etc. Then the original pointer program is replaced by an integer program updating these numerical measures. The new numerical program is constructed in such a way that it simulates (or bisimulates) the original pointer program.

[7] Appendix E

In [BBH+06, BFL06], the authors showed that pointer programs manipulating singly linked lists are bisimular to counter automata. The idea is based on the fact that a number of uninterrupted SLL segments is linear to the number of pointer variables. Up to the data values, each heap can be represented by remembering the interconnection of its SLL segments and the lengths of the particular segments. The number of possible interconnections of SLL segments is finite, hence each interconnection is represented as a state in the finite control of CA. The lengths of particular segments are represented by counters. The verification tools for CA can be easily applied.

In [HIRV07], we proposed an encoding of programs manipulating trees into CA based on the following numerical measures—for each pointer variable, one assigns two counters representing the size (the number of nodes), respective the height of the tree pointed by the variable. The constructed CA then simulates the original pointer program hence reachability (resp. termination) proof for the CA implies reachability (resp. termination) proof for the pointer program.

The most general technique is the one proposed in the THOR tool [MTLT08]. The numerical measures are not fixed for the concrete type of data structures, and the user can define its own data structures together with the numerical measures. The data structures are defined in separation logic (c.f. Section 3.2) extended with numerical constraints. The authors demonstrated their approach on DLLs and trees and implemented it in a tool called THOR. However, to the best of our knowledge, the implementation of THOR has hard-wired DLL definition and other type of data structures cannot be easily added.

2.4 Graph Automata

Now, we proceed to notion of graph automata mentioned in the work [MP11][8]. We consider this notion interesting, because it allows one to describe wide class of graph languages as, e.g., a language of all 3-colorable graphs (see below). Despite the notion of graph automata sounds promising, it is not that useful for verification because there are no known efficient algorithms for language inclusion and emptiness problems.

Let us consider a class $\mathcal{C}$ of oriented edge-labeled graphs and let Σ be a finite set of edge labels (the alphabet). A *graph automaton* (GA) over Σ is a triple $A = (Q, \{T_a\}_{a\in\Sigma}, type)$, where

- Q is a finite set of states,
- $T_a \subseteq Q \times Q$ is a *tiling relation*,
- $type : Q \to 2^\Sigma \times 2^\Sigma$ is a *type relation*.

A graph is accepted by a GA iff each node of the graph can be labeled by a state such that (i) each edge satisfies the tiling relation and (ii) the incoming and outgoing edges of each node conform to the type relation. Formally, a graph $G = (V, \{E_a\}_{a\in\Sigma})$ is accepted by the automaton G if there exists a mapping $\rho : V \to Q$ such that the following holds:

[8]The version used in [Tho91] differs in some details, but the main spirit is equal.

- $\forall (v_1, v_2) \in E_a : (\rho(v_1), \rho(v_2)) \in T_a$
- $\forall v \in V : type(\rho(v)) = (In, Out)$, where $In = \{a | \exists v'.(v', v) \in E_a\}$ and $Out = \{a | \exists v'.(v, v') \in E_a\}$.

Example: Let us fix $\Sigma = \{x\}$. The automaton $A = (\{q_1, q_2, q_3\}, \{T_x = \{(q, r) \mid q \neq r\}, type\}$ where $type(q) = (\{x\}, \{x\})$ for each $q \in \{q_1, q_2, q_3\}$ accepts all 3-colorable graphs where each node has at least one incoming and one outgoing edge. □

Closure Properties and Decidable Problems

The class of graph automata is closed under union and intersection, but it is not closed under complement. Moreover, the inclusion and emptiness problems are not decidable in general.

The inclusion and emptiness become decidable, when one restricts the class of graphs to the graphs with bounded tree width [MP11]. The decidability result is then based on a reduction from graph automata to MSO on graphs (see Section 3.1) which leads to a $NONELEMENTARY$ decision procedure. This decision procedure is a theoretical one and we are not aware of any heuristic technique targeting these problems for GA. Therefore, in the current state of the art, GA can not be practically applied within verification.

Chapter 3

Logic-based Techniques for Analysis of Pointer Programs

Various kinds of logics represent a popular way to encode potentially infinite sets of elements. In the case of shape graphs, one of the most general is MSO logic on graphs. But, it is not clear how to symbolically execute program statements on the level of MSO on graphs (c.f. Section 3.1). Moreover, MSO on graphs is undecidable in general. Therefore more specialized logic approaches are in use when aiming at an efficient analysis of pointer manipulating programs. In the last 10 years, the most popular logic is probably separation logic (c.f. Sec. 3.2).

3.1 Monadic Second-Order Logic

Monadic second-order logic (MSO) is a powerful logic, which can be interpreted over various domains. In this section, we provide a brief introduction to MSO interpreted on strings, trees, and graphs. MSO logics on strings and trees play important role because they are decidable, and, despite the worst case complexity of the decision procedures, efficient heuristic-based algorithms are implemented in the state-of-the-art tools.

The syntax of MSO is based on the following components:

- *First order variables* VAR_1: A first order variable is a logical representation of a single element, such as a position in a string, word, or tree; or a vertex in a graph. First order variables are denoted by small letters ($x, y, z, \ldots$).
- *Second order variables* VAR_2: A second order variable is a logical representation of a (potentially empty) set of elements, such as a set of positions in a string or tree; or a set of vertices. Second order variables are denoted by capital letters ($X, Y, Z, \ldots$).
- *Quantifiers* $\exists x, \forall x, \exists X, \forall X$: Each $x \in VAR_1$ and $X \in VAR_2$ may be quantified existentially or universally.
- *Predicate* $A(x)$: for $x \in VAR_1$ and $A \in VAR_2$: $A(x)$ is valid iff $x \in A$.

- Other predicate symbols depending on the domain on which the MSO is interpreted.

- *Standard logical connectives*: $\wedge, \vee, \neg, \rightarrow$

MSO on Strings and Trees

In order to interpret MSO on *strings*, one needs to add a predicate $succ(x, y)$, which states that *x and y are successor positions in the string*. The expressive power of MSO interpreted over *strings* (MSO_S) is equal to Büchi automata [Büc60].

If the quantification of second-order variables is restricted to finite sets only, then the logic is called *weak*. Weak MSO on strings is often denoted as WS1S (*Weak Second order theory of One Successor*), and its power is equal to finite automata. See, e.g., [Tho90, KM01] for more details.

To interpret MSO on k-ary *trees*, a set of predicates $\{succ_i(x, y) \mid 0 < i \leq k\}$ must be added, where $succ_i(x, y)$ states that *the position y is the i^{th} son of the position x*. As in the case of strings, the theory is called *weak* if the quantification of second-order variables is restricted to finite sets only. WSkS then denotes a Weak monadic Second order theory of k Successors. The power of WSkS is equivalent to finite tree automata [Tho90].

Automata-Logic Connection

As we have already mention above, for the string case as well as the tree case, the WSkS logic is tightly connected to finite (tree) automata. For a WSkS formula ϕ, one can efficiently create a finite (tree) automaton A_ϕ such that $w \models \phi \Leftrightarrow w \in \mathcal{L}(A_\phi)$ and also an automaton A can be translated into an equivalent formula ϕ_A. Within the translation, subsets of second order variables are mapped to alphabet symbols, $\vee$ is mapped to language union, $\wedge$ to language intersection, $\neg$ to complement, etc. This translation leads to a decision procedure for WSkS, which belongs in general to the $NONELEMENTARY$ complexity class. For a given formula ϕ with k quantifier alternations, the decision procedure has $DTIME(\underbrace{2^{2^{\cdots^{2^{(|\phi|)}}}}}_{k+1})$ worst case complexity.

The automata-based decision procedure was, with many optimizations, for the first time implemented in the MONA tool [KM01]. MONA uses a binary encoding of alphabet symbols, a semi-symbolic encoding of deterministic finite (tree) automata using BDDs [Bry92], and a number of other optimizations to make the decision procedure tractable. Last year, a novel technique based on anti-chains was implemented in the tool dWiNA [FHLV15], where the state-of-the art algorithms for nondeterministic automata were employed to avoid the $EXPTIME$ worst case complexity of the determinization in a lot of practical cases and subsequently increase practical limits of the decision procedure for WSkS. The anti-chain-based technique was recently significantly improved by a number of heuristics [FHJ+16].

PALE: Verification Based on MSO on Trees

Here, we shortly introduce the Pointer Assertion Logic Engine (PALE) [MS01] as an example of successful application of MSO on trees in verification. PALE uses an approach for encoding a set of shape graphs as a set of trees using Graph types [KS93]. As was already mentioned in Sec. 2.2, graph types represent a data structure by a tree together with extra non-tree edges. The non-tree edges are represented by means of so-called routing expressions placed in source nodes. A (possibly infinite) set of trees with routing expression can be encoded as a formula in MSO interpreted on trees. The tree backbone cannot be easily changed within verification because this would invalidate all the routing expression. Therefore each destructive pointer update such as $x \to s = y$ is performed by a change of the corresponding routing expressions only. This allows one to express an effect of a finite block of code with no loops in MSO. Together with user provided invariants for each loop and each procedure call and exit, PALE is able to prove correctness of programs by using MONA [KM01] as its backend.

MSO on Graphs

MSO can be interpreted on graphs as well. The needed component is a set of predicates $\{edge_a(x, y) | a \in \Sigma_e\}$, where Σ_e is a set of edge labels. The predicate $edge_a(x, y)$ is valid, if there is an a-labeled edge between the nodes x and y.

MSO interpreted on graphs is in general undecidable. Nevertheless, for each $k \in \mathbb{N}$, MSO interpreted over the class of graphs restricted to the tree width of at most k is decidable. The proof of decidability is based on a reduction of formula ϕ_G in MSO on graphs to a formula ϕ_T in MSO on trees such that $\models^k \phi_G \Leftrightarrow \models \phi_T$. The reduction adds a small fixed number of quantifier alternations to the number of quantifier alternations occurring in ϕ_G and leads to a $NONELEMENTARY$ decision procedure—see the Courselle theorem [FG06] for more details.

Potential Problems of Verification Based on MSO on Graphs

The idea of using MSO interpreted on graphs of bounded tree width as a symbolic model for shape graphs seems to be nice on the first view. One can encode sets of graphs as, e.g., singly/doubly linked lists of any length, trees (with parent or root pointers), trees with linked leaves, etc. But there are few drawbacks which make MSO on graphs non-suitable for the purpose of verification. The main problem is related to execution of destructive updates as, e.g., $x \to s = y$. In order to interpret this statement, one needs to disconnect the original s-labeled edge from the node x and connect it to the node y, and, moreover, this must be done on a set of graphs. But one cannot simply remove a predicate $edge_s(x, z)$ from the original formula because some other edges can be defined using paths in the graph going via the $edge_s(x, z)$—a removal of a single edge can break an unbounded number of other edges.

Example: Let us consider the conjunction of the following 4 formulae describing a set of all possible SLLs (Singly-linked lists) where the pointer variable $head$ points to the

first element of the list, $tail$ points to the last element of the list and x points somewhere to the middle.

1. $\exists LIST.\, x \in LIST$
2. $\forall e \in LIST.\, e = head \vee$
   $(\exists y \in LIST.\, edge_n(y, e) \wedge \forall z \in LIST.\, edge_n(z, e) \rightarrow z = y)$
3. $tail \in LIST \wedge edge_n(tail, null)$
4. $\forall X.(head \in X \wedge \forall y \in X.edge(y, q) \rightarrow q = null \vee q \in X) \rightarrow LIST \subseteq X$

Condition (1) guarantees that x points somewhere to the list, (2) guarantees that each element of the list has unique predecessor or it is equal to $head$, (3) guarantees that $tail$ is the last element of the list, and condition (4) guarantee that everything is reachable from head—LIST is the smallest set satisfying the conditions above. A potential execution of the statement $x \rightarrow next = y$ leads to a need of global change of this formula. □

The second drawback of MSO on graphs is that within verification, one often needs to decide validity of implication to check a fixpoint condition. Due to the $NON-ELEMENTARY$ worst case, one has to employ some (heuristic-based) efficient algorithms to make decision procedure for MSO on graphs tractable, and, to the best of our knowledge, such efficient algorithms does not exists in current state of the art. On the other hand, MSO on graphs can be used within theoretical decidability proofs of other symbolic models such as separation logic (c.f. Section 3.2) or forest automata (c.f. Section 2.2).

3.2 Separation Logic

Separation logic (SL) [Rey02] is a widely used symbolic representation of dynamic data structures generated by pointer programs. In the past years, several academic tools (e.g. *Space Invader* [BCC$^+$07], *Sleek* [NC08], or *Predator* [DPV11]) as well as industrial tools (e.g. Infer [CD11]) were built on top of SL or were based on its main ideas.

The crucial principle of SL is the principle of *local reasoning*. The main idea of the local reasoning can be characterized as follows: A local change in a small piece of memory is mapped to a local change in a small subformula of the SL formula. This allows ones easily analyze a single function or thread. The analyzer considers only a small subformula corresponding to the heap accessed by the function (resp. thread) and then results of different functions (resp. threads) of the program are combined.

The basic syntax of SL consist of the following components:

- The empty heap emp.
- The points-to predicate $x \mapsto (s_1, \ldots, s_k)$, which denotes the fact that there is an allocated memory cell labeled by the logical variable x with pointer references to cells labeled by the logical variables $s_1, \ldots, s_k$.

- The separation conjunction $*$. Formula $\phi * \sigma$ denotes the fact that the set of allocated cells in ϕ and in σ are disjoint (they may be connected by the pointer references only).

These basic components can then be connected using standard logical connectives ($\vee, \wedge, \neg, \dots$), the logical variables may be quantified existentially or universally. The program variables are represented by free variables. For more details in syntax and semantics, see, e.g., [Rey02].

Representation of Infinite Sets by SL

The basic notion of SL allows one to describe a finite set of finite heaps. But, in verification, one often needs to represent infinite sets of finite memory configurations as, e.g., *all possible doubly-linked lists*. Therefore the so-called *recursive definitions* (also called *recursive predicates*) were introduced into SL. A recursive predicate $P(x_1, \dots, x_n)$ may be defined as a disjunction of rules, where each rule is a SL formula, which may contain some other recursive predicate calls including P itself. SL with recursive definitions is denoted as SLRD in the following. Several examples of recursive definitions are depicted in Fig. 3.1.

$$
\begin{array}{rcl}
list(hd, tl) & ::= & emp \wedge hd = tl \\
 & | & \exists x.\ hd \mapsto x * list(x, tl) \\
dll(hd, p, tl) & ::= & emp \wedge hd = tl \\
 & | & \exists x.\ hd \mapsto (x, p) * dll(x, hd, tl) \\
tree(root) & ::= & emp \wedge root = nil \\
 & | & \exists l, r.\ root \mapsto (l, r) * tree(l) * tree(r) \\
tll(x, ll, lr) & ::= & x \mapsto (nil, nil, lr) \wedge x = ll \\
 & | & \exists l, r, z.\ x \mapsto (l, r, nil) * tll(l, ll, z) \\
 & & *tll(r, z, lr)
\end{array}
$$

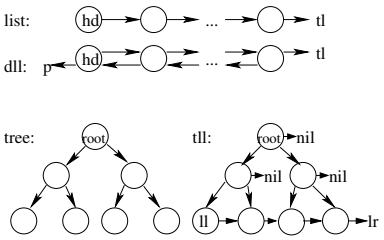


Figure 3.1: Examples of recursive data structures definable in SLRD.

Entailment Problem in Separation Logic

In order to use SLRD in verification, one of the most important problems to be solved is the so-called *entailment problem*. Given two formulae ϕ and ψ, the problem of entailment $\phi \stackrel{?}{\models} \psi$ asks whether each model of formula ϕ is also a model of ψ.

The entailment problem is in general undecidable. But there are plenty of works based on various restriction of recursive definitions in order to obtain a decidable (or semi-decidable) fragment of SLRD.

- *Linked lists*: A set of works proved decidability of SLRD restricted to linked lists. The work [CHO+11] provides a PTIME decision procedure for SL with a hard coded *SLL* predicate. This work has been generalized to nested and overlayed lists

[ESS13] too. Recently, a set of works provided reduction from SL with SLL/DLL recursive definitions into SMT [PWZ13, PR13], which allows one to take advantage of current SMT solvers. The entailment problem for SL with various kinds on nested linked lists was also considered in the tool SPEN [ELSV14], which was later extended also to trees for the price of completeness [ESW15].

- *Trees:* The problem of sound and complete decision procedure for SLRD restricted to tree-like structures was tackled in our work [IRV14][1], where the SLRD is translated into tree automata and hence the entailment problem is reduced into language inclusion problem. This reduction provides an $EXPTIME$ upper bound. In case of non-tree data structures, the same reduction provides a sound approximation of the entailment. Another work [PWZ14] proposed a specialized logic GRIT for trees and its $NPTIME$ decision procedure based on reduction into SMT. SL restricted to trees can be easily translated into this logic.

- *General structures:* A general decidability result for SLRD (with some necessary restrictions) was provided in our original work [IRŠ13][2]. The result is based on a translation of a formula ϕ in SLRD into a formula ϕ_{MSO} in MSO on Graphs (c.f. Section 3.1). The decision procedure for MSO on graphs can then be employed. This result shows theoretical decidability bounds for SLRD because the decision procedure for MSO on graphs is in general $NONELEMENTARY$.

- *Semi-decision procedures:* SLEEK [NC08] implements an entailment check based on unfoldings and unifications. The technique behind SLEEK is sound, but its termination is not guaranteed even if the entailment holds. Another semi-algorithmic approach is implemented in the tool CYCLIST [BGP12], where two semi-algorithms are provided—one for proving entailment and one for its disproving [BG15].

Verification with SL

From the beginning, SL was designed as a suitable formalism for verification. Due to the local reasoning and the separation conjunction, a destructive update $x \to s = y$ can be performed simply by changing the points-to predicate $x \to (\underbrace{\dots, z}_{s}, \dots)$ to a new predicate $x \to (\underbrace{\dots, y}_{s}, \dots)$ with no influence to the rest of the heap.

However, when recursive definitions are used, one needs to implement fold and unfold operations and some variant of the entailment check. The unfold operation is needed, when a selector hidden inside a predicate is accessed. The more critical is the fold operation, which is needed to compute sets of shape graphs reachable at exit points of unbounded loops. Fold is often combined with overapproximation of the current heap. There is no generic algorithm, so several heuristics are used to tackle the fold operation.

[1] Appendix F
[2] Appendix D

3.3 Other Logics

TVLA: TVLA (the *Three Valued Logic Analyzer*) [SRW02] uses a symbolic representation of heap based on the Kleene 3-valued interpretation [Kle87] of predicate logic with transitive closure over graphs. To encode a set of heaps, a principle of *summary nodes* is used. Summary nodes are used for the same reason as *boxes* in forest automata (c.f. Section 2.2) or recursive definitions in separation logic—to represent a complicated shape graph, or a possibly infinite set of shape graphs, by a single element. A concretization operation is used to access inner nodes and summarization is used for acceleration of verification.

STRAND: STRAND [MPQ11] is a logic that allows one to reason about heap and data. It is based on MSO interpreted on trees enhanced by data constrains. The formulas are of the form $\exists \overrightarrow{x} \forall \overrightarrow{y}\, \varphi$, where φ is an MSO formula over trees (possibly containing more quantifiers) with additional data constrains. The data constrains can be only over the variables from $\overrightarrow{x}$ and $\overrightarrow{y}$. A combination of $MONA$ [KM01] and the $Z3$ SMT solver [MB08] is then used to decide the logic.

Alias Logic: Alias logic [BIL03] presents an other possibility to reason about pointer programs. The logic is defined on a storeless memory model—a memory configuration is not represented as a graph. Instead, each heap node is described by the regular language of paths going to it through the heap. This language is then represented as a finite automaton. Alias logic then allows one to express regular properties of the heap.

Chapter 4

Summary of Contributions

This habilitation thesis comprises a collection of journal and conference research papers. For each paper in the collection, the author of the thesis has contributed in a significant of key way to the development of concepts behind the paper as well as to the writing of the papers themselves. Moreover, for the papers included as Appendices A, F, and G, the author of the thesis is also the author of the implementation used for the experimental evaluation presented in the particular papers. The papers in the collection are ordered by the date of their first publication.

Automata-Based Termination Proofs—Appendix A

In this paper, we tackle the termination problem of pointer programs. We proposed an original abstraction of these programs into Büchi automata and reduced the termination problem to the language intersection problem. As an underlying technique, we use our previous encoding of pointer programs into tree automata with routing expressions (c.f. Sec. 2.2).

This paper was originally published in the *Proceedings of International Conference Implementation and Application of Automata* (CIAA) in the year 2009, and later on, its extended version, attached as appendix A, was published in the *Computing and Informatics* (CAI) journal in the year 2013.

Forest Automata for Verification of Heap Manipulation—Appendix B

In this paper, we proposed the forest encoding of memory configurations of pointer programs together with the principles of forest automata and boxes (c.f. Sec. 2.2). This encoding was employed for a symbolic verification and implemented in the prototype tool Forester. The technique is fully automated up to the fact that suitable boxes must be provided manually[1].

This paper was originally published in the *Proceedings of International Conference on Computer Aided Verification* (CAV) in the year 2011. The version attached as Appendix B was published in the journal of *Formal Methods in System Design* in the year 2012 for

[1] Boxes for most common data structures are included in the implementation.

which the paper was chosen by the program committee of CAV conference. By now, this work (the conference together with the journal version) was cited by 5 other papers.

Abstract Regular (Tree) Model Checking—Appendix C

This work is a summary journal paper published in a special issue of the *International Journal on Software Tools for Technology Transfer* (STTT) devoted to regular model checking. The paper summarizes a number of contributions in the area of abstract regular model checking, out of which a number was proposed by the author of the thesis. By now, this work was cited by 9 other works.

The Tree Width of Separation Logic with Recursive Definitions—Appendix D

In this paper, we proved that any separation logic formula (c.f. Section 3.2) using rather general recursively defined predicates is decidable for satisfiability, and, moreover, entailments between such formulae are decidable for validity. These predicates are general enough to define (doubly-) linked lists, trees, and structures more general than trees, such as trees whose leaves are chained in a list. The decidability proof is based on a reduction to MSO on graphs with bounded tree width.

This paper was published in the *Proceedings of International Conference on Automated Deduction* (CADE-24) in the year 2013, and it received the best paper award for *proving decidability of satisfiability and entailment problem in a fragment of separation logic and closing in a creative way a problem open since 2004*. By now, this work was cited by 26 other works.

Fully Automated Shape Analysis Based on Forest Automata—Appendix E

This paper is a follow-up work on the paper *Forest Automata for Verification of Heap Manipulation*—Appendix B. We proposed an automated learning of the boxes (see Section 2.2) and increased the level of automation of the forest automata based verification.

This paper was published in the *Proceedings of International Conference on Computer Aided Verification* (CAV) in the year 2013. By now, this work was cited by 5 other papers.

Deciding Entailments in Inductive Separation Logic with TA—Appendix F

This work is a continuation of our work on separation logic. In this work, we reduce the entailment problem for a non-trivial subset of separation logic (c.f. Section 3.2) describing trees (and beyond) to the language inclusion of tree automata. Our reduction provides tight complexity bounds for the problem and shows that entailment in our fragment is EXPTIME-complete.

The paper was published in the *Proceeding of International Symposium on Automated Technology for Verification and Analysis* (ATVA) in the year 2014. By now, this work was cited by 3 other papers.

Abstraction Refinement and Antichains for Trace Inclusion of Infinite State Systems—Appendix G

In this work, we tackle the trace inclusion problem of data automata. A *data automaton* is a finite automaton equipped with variables (counters or registers) ranging over infinite data domains. A trace of a data automaton is an alternating sequence of alphabet symbols and values taken by the counters during an execution of the automaton. Since the problem is undecidable in general, we give a semi-algorithm based on abstraction refinement, which is proved to be sound and complete modulo termination. We plan to use this work as an underlying technique for the entailment problem in separation logic (c.f. Section 3.2) equipped by data values placed in allocated nodes.

This work was published in the *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS) in the year 2016.

Chapter 5

Conclusions and Future Work

This thesis summarizes the current state of the art in the area of symbolic representations of memory configurations of programs with pointers. We focused especially on the representations based on various kinds of automata and logics. In the area of automata-based representations, we mostly targeted techniques based on finite word and tree automata. These techniques can be easily used within the framework of regular model checking to compute sets of reachable states. In the area of logics, the thesis mostly targeted monadic second order logic and separation logic. We discussed decidability issues of these logics and possibilities of symbolic verification based on them. The thesis is based on a collection of papers where the author of the thesis is a key contributor of each particular one.

Despite the huge progress in the last years, there is still a lot of open questions in the area of the thesis. The main goal of future work is to establish a fully-automated scalable technique for production software. To achieve this goal, one can divide the future work into several categories.

Symbolic encoding for real-life low-level code: The existing symbolic representations are already quite general, but when one starts to deal with a real life low-level code, one would find a lot of intricate constructs related to pointer arithmetics (see, e.g., Linux Kernel DLL implementation [DPV13]), and the symbolic verification must be capable to deal with it. There are already some works targeting pointer programs together with pointer arithmetics, but there is still a lot of open space for future improvements, especially when one needs to deal with complex data structures.

Another problem with the real code is that its correctness often depends on a combination of several domains as, e.g., pointer manipulations together with integer values or arrays. A lot of work was done for these particular domains, but one needs a combined symbolic representation to tackle correctness of the code based on a combination of domains. Here, we plan to employ our work on trace inclusion [IRV16][1] as an backend to separation logic combined with integer values.

[1] Appendix G

Scalability: A lot of currently used verification techniques cannot handle large software projects. The problem is mostly related to the fact that verification must start from the *main* function and construct the invariants for the whole project. Therefore some works based on so-called *bi-abduction* [CDOY09, LGQC14] were proposed to tackle the problem with scalability. These techniques can compute invariants for each particular function separately and then compare whether the results match together. The open problem is to establish the bi-abduction techniques for some automata-based symbolic domain with a goal to benefit from advantages of automata-based verification and bi-abduction together.

Decision procedures: Decision procedures for symbolic encodings are an integral part of logic-based verification. They are quite established for linked lists and trees, but there is still huge gap in the area of generic data structures—generic techniques are often not tractable and heuristics often fail. Together with the automated learning of suitable symbolic containers such as recursive predicates, the simple list-like approaches stop to work. The reason is that the learned containers can be rather tricky and their comparison is not straightforward. Moreover, when one starts to combine pointer programs with other domains, one needs to combine various decision procedures into a new one as well.

Termination and bound analysis: Up to some preliminary results, most of the work targeting verification of pointer programs tackle the problem of safety analysis. But an important question is also (i) whether the program terminates for all inputs and (ii) how long does it take to terminate. These kind of techniques are usually based on providing so-call norms over data structures or their symbolic representations. Such a norm can be, e.g., the length of the list, height of the tree, number of nodes, etc. With suitable norms, one can translate a pointer program into an arithmetic one (as e.g. in [BBH$^+$06, BFL06, HIRV07]) and then use some technique for termination or bound analysis of arithmetic programs [Ryb, CPR06, BIKV, SZ10]. A problem with this approach is usually related to the fact that one has to first construct invariants over some symbolic representation and then derive the norms. But within the invariant construction, one usually applies some overapproximation, which may be suitable for safety analysis, but not strong enough to prove termination or even derive some bounds.

Bibliography

[Abd12] P. A. Abdulla. Regular model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):109–118, 2012.

[AO08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[BBH+06] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *Proc. of CAV'2006*, volume 4144 of *LNCS*. Springer, 2006.

[BBLS00] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S Systems to Verify Prameterized Networks. In *Proc. of TACAS 2000*, volume 1785 of *LNCS*. Springer, 2000.

[BCC+07] 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. CAV'07*, volume 4590 of *LNCS*. Springer, 2007.

[BCK01] P. Baldan, A. Corradini, and B. König. A Static Analysis Technique for Graph Transformation Systems. In *Proc. of CONCUR'01*, volume 2154 of *LNCS*. Springer, 2001.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[BFL06] S. Bardin, A. Finkel, and E. Lozes. From Pointer Systems to Counter Systems Using Shape Analysis. In *Proc. of AVIS'06*, 2006.

[BG15] J. Brotherston and N. Gorogiannis. Disproving inductive entailments in separation logic via base pair approximation. In *Proc. of TABLEAUX-24*, volume 9323 of *LNAI*, pages 287–303. Springer, 2015.

[BGP12] J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *Proc. of APLAS'12*, volume 7705 of *LNCS*. Springer, 2012.

[BHMV05] 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*, volume 3440 of *LNCS*. Springer, 2005.

[BHRV05] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. In *Proc. of Infinity'05*, 2005.

[BHRV06] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

[BHRV12] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[BIKV] M. Bozga, R. Iosif, F. Konečný, and T. Vojnar. Flata.
URL: `http://nts.imag.fr/index.php/Flata`.

[BIL03] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[BLBS01] K. Baukus, Y. Lakhnech, S. Bensalem, and K. Stahl. Networks of Processes with Parameterized State Space. In *Proc. of VEPAS'01*, volume 50 of *ENTCS*. Elsevier, 2001.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[BT12] A. Bouajjani and T. Touili. Widening techniques for regular tree model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):145–165, 2012.

[Büc60] J. R. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Zeitschr. f. Math, Logik und Grundlagen d. Math*, 6:66–92, 1960.

[CD11] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of c programs. In *Proc. of NFM'11*, volume 6617 of *LNCS*. Springer, 2011.

[CDOY09] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proc. of POPL*. ACM, 2009.

[ČEV07a] M. Češka, P. Erlebach, and T. Vojnar. Generalised multi-pattern-based verification of programs with linear linked structures. *Formal Aspects of Computing*, 19(3):363–374, 2007.

[ČEV07b] M. Češka, P. Erlebach, and T. Vojnar. Pattern-based verification for trees. In *Proc. of EUROCAST'07*, volume 4739 of *LNCS*. Springer, 2007.

[CHO+11] B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *Proc. of CONCUR'11*, volume 6901 of *LNCS*. Springer, 2011.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In *Proc. of CAV 2006*, volume 4144 of *LNCS*. Springer, 2006.

[CR00] S.J. Creese and A.W. Roscoe. Data Independent Induction over Structured Networks. In *Proc. of PDPTA 2000*. CSREA Press, 2000.

[CR07] Sigmund Cherem and Radu Rugina. Maintaining Doubly-Linked List Invariants in Shape Analysis with Local Reasoning. In *Proc. of VMCAI'07*, volume 4349 of *LNCS*. Springer, 2007.

[DEG06] J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[Deu94] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. In *Proc. of PLDI'94*. ACM Press, 1994.

[DPV11] 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*, volume 6806 of *LNCS*. Springer, 2011.

[DPV13] K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. In *Proc. of SAS'13*, volume 7935 of *LNCS*. Springer, 2013.

[ELSV14] C. Enea, O. Lengál, M. Sighireanu, and T. Vojnar. Compositional entailment checking for a fragment of separation logic. In *Proc. of APLAS'14*, volume 8858 of *LNCS*. Springer, 2014.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.

[ESS13] C. Enea, V. Saveluc, and M. Sighireanu. Compositional invariant checking for overlaid and nested linked lists. In *Proc. of ESOP'13*, volume 7792 of *LNCS*. Springer, 2013.

[ESW15] C. Enea, , M. Sighireanu, and Z. Wu. Automating Program Proofs Based on Separation Logic with Inductive Definitions. Technical Report TR-20150209, LIAFA, Université Paris Diderot, France, 2015.

[FG06] J. Flum and M. Grohe. Tree width. In *Parameterized Complexity Theory*, Texts in Theoretical Computer Science. An EATCS Series, pages 261–299. Springer Berlin Heidelberg, 2006.

[FHJ+16] T. Fiedor, L. Holík, P. Janků, O. Lengál, and T. Vojnar. Lazy automata tehcniques for ws1s. Technical Report FIT-TR-2016-01, FIT BUT, Czech republic, 2016.

[FHLV15] T. Fiedor, L. Holík, O. Lengál, and T. Vojnar. Nested antichains for ws1s. In *Proc. of TACAS'2015*, volume 9035 of *LNCS*. Springer, 2015.

[FKLV12] J. Fiedor, B. Křena, Z. Letko, and T. Vojnar. A uniform classification of common concurrency errors. In *Proc. of EUROCAST'01*, volume 6927 of *LNCS*. Springer, 2012.

[GS92] S.M. German and A.P. Sistla. Reasoning about Systems with Many Processes. *Journal of the ACM*, 39(3):675–735, 1992.

[GS97] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[HHR+11] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest automata for verification of heap manipulation. In *Proc. of CAV'2011*, volume 6806, 2011.

[HIRV07] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. In *Proc. of ATVA'07*, volume 4762 of *LNCS*. Springer, 2007.

[HJKN15] J. Heinen, C. Jansen, J.P. Katoen, and T. Noll. Juggrnaut: using graph grammars for abstracting unbounded heap structures. *Formal Methods in System Design*, 47(2):159–203, 2015.

[HLR+13] L. Holík, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. Fully automated shape analysis based on forest automata. In *Proc. of CAV'13*, number 8044 in LNCS. Springer, 2013.

[IR13] R. Iosif and A. Rogalewicz. Automata-based termination proofs. *Computing and Informatics*, 2013(4):739–775, 2013.

[IRŠ13] R. Iosif, A. Rogalewicz, and J. Šimáček. The tree width of separation logic with recursive definitions. In *Proc. of CADE-24*, volume 7898 of *LNCS*. Springer, 2013.

[IRV14] R. Iosif, A. Rogalewicz, and T. Vojnar. Deciding entailments in inductive separation logic with tree automata. In *Proc. of ATVA'14*, volume 8837 of *LNCS*. Springer, 2014.

[IRV16] R. Iosif, A. Rogalewicz, and T. Vojnar. Abstraction refinement and antichains for trace inclusion of infinite state systems. In *Proc. of TACAS'16 (to appear)*, LNCS. Springer, 2016.

[Jon81] H.B.M. Jonkers. Abstract Storage Structures. In *Algorithmic Languages*. IFIP, 1981.

[Kle87] S. Kleene. *Introduction to Mathematics*. North-Holland, 1987.

[KM95] R.P. Kurshan and K.L. McMillan. A Structural Induction Theorem for Processes. *Information and Computation*, 117(1), 1995.

[KM01] N. Klarlund and A. Møller. MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. *Theoretical Computer Science*, 256(1–2), 2001.

[KS93] N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*. ACM Press, 1993.

[LAIS06] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for Shape Analysis with Fast and Precise Transformers. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[Leg12] A. Legay. Extrapolating (omega-)regular model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):119–143, 2012.

[Let12] Z. Letko. *Analysis and Testing of Concurrent Programs*. PhD thesis, Faculty of Information Technology BUT, 2012.

[LGQC14] Q. L. Le, C. Gherghina, S. Qin, and W. N. Chin. Shape analysis via second-order bi-abduction. In *Proc. of CAV'14*, volume 8559 of *LNCS*, 2014.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic Verification of Parameterized Linear Networks of Processes. In *Proc. of POPL'97*. ACM Press, 1997.

[LYY05] O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proc. of ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.

[MB08] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.

[McM06] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.

[MP11] P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *Proc. of POPL'11*. ACM, 2011.

[MPQ11] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *Proc. of POPL'2011*. ACM, 2011.

[MQS00] K.L. McMillan, S. Qadeer, and J.B. Saxe. Induction in Compositional Model Checking. In *Proc. of CAV 2000*, volume 1855 of *LNCS*. Springer, 2000.

[MS01] A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001. Also in SIGPLAN Notices 36(5), 2001.

[MTLT08] S. Magill, M. Tsai, P. Lee, and Y. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *Proc. of CAV*, volume 5123 of *LNCS*. Springer, 2008.

[NC08] H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *Proc of CAV'08*, volume 5123 of *LNCS*. Springer, 2008.

[PR13] J. A. N. Pérez and A. Rybalchenko. Separation logic modulo theories. In *Proc. of APLAS'13*, volume 8301 of *LNCS*. Springer, 2013.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic Deductive Verification with Invisible Invariants. In *Proc. of TACAS'01*, volume 2031 of *LNCS*. Springer, 2001.

[PWZ13] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In *Proc. of CAV'13*, volume 8044 of *LNCS*. Springer, 2013.

[PWZ14] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. In *Proc. of CAV'14*, volume 8559 of *LNCS*. Springer, 2014.

[Rey02] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.

[Ryb] A. Rybalchenko. ARMC: Abstraction Refinement Model Checker. URL: `http://www.mpi-inf.mpg.de/~rybal/armc/`.

[SRW02] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.

[SZ10] M. Sinn and F. Zuleger. LOOPUS - A tool for computing loop bounds for C programs. In *Proc. of WING'09*, volume 1 of *EPiC Series*. EasyChair, 2010.

[Tho90] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science (Vol. B)*, pages 133–191. MIT Press, 1990.

[Tho91] W. Thomas. On logics, tilings, and automata. In *Proc. of Automata, Languages and Programming*, volume 510 of *LNCS*. Springer, 1991.

[Ven99] A. Venet. Automatic Analysis of Pointer Aliasing for Untyped Programs. *Science of Computer Programming*, 35(2), 1999.

[WKZ+06] T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On Verifying Complex Properties Using Symbolic Shape Analysis. Technical Report MPI-I-2006-2-1, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2006.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In *Proc. of Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer, 1989.

[YKB02] T. Yavuz-Kahveci and T. Bultan. Automated Verification of Concurrent Linked Lists with Counters. In *Proc. of SAS'02*, volume 2477 of *LNCS*. Springer, 2002.

Appendix A

Automata-Based Termination Proofs

Computing and Informatics, Vol. 32, 2013, 739–775

AUTOMATA-BASED TERMINATION PROOFS

Radu Iosif

VERIMAG, CNRS
2 av. de Vignate
38610 Gières, France
e-mail: iosif@imag.fr

Adam Rogalewicz

FIT, Brno University of Technology
IT4Innovations Centre of Excellence
Božetěchova 2
612 66 Brno, Czech Republic
e-mail: rogalew@fit.vutbr.cz

Communicated by Jacek Kitowski

Abstract. This paper describes our generic framework for detecting termination of programs handling infinite and complex data domains, such as pointer structures. The framework is based on a counterexample-driven abstraction refinement loop. We have instantiated the framework for programs handling tree-like data structures, which allowed us to prove automatically termination of programs such as the depth-first tree traversal, the Deutsch-Schorr-Waite tree traversal, or the linking leaves algorithm.

Keywords: Formal verification, termination analysis, Büchi automata, tree automata, programs with pointers

Mathematics Subject Classification 2010: 68N30, 68Q60, 68Q70, 68W30

1 INTRODUCTION

Proving termination is an important challenge for the existing software verification tools, requiring specific analysis techniques [26, 8, 24]. The basic principle underlying all these methods is proving that, in every (potentially) infinite computation of the program, there exists a suitable measure which decreases infinitely often. The commonly used measures are from so-called well-founded domains. The well-foundedness guarantees that in the domain there is no infinite decreasing sequence. As a consequence, the measure cannot decrease infinitely often, and hence the program must terminate.

In this paper, we provide an extended and improved description of our termination analysis, originally presented in [21]. The proposed termination analysis is based on the following principles:

1. We consider programs working on infinite data domains $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ equipped with an arbitrary number of well-founded partial orders.
2. For any transformation $\Rightarrow \subseteq D \times D$ induced by a program statement, and any partial order on D $\preceq_i$, $1 \leq i \leq n$, we assume that the problem $\Rightarrow \cap \preceq_i \stackrel{?}{=} \emptyset$ is decidable algorithmically.
3. An abstraction of the program is built automatically and checked for the existence of potential non-terminating execution paths. If such a path exists, then an infinite path of the form $\sigma\lambda^\omega$ (called *lasso*) is exhibited.
4. Due to the over-approximation involved in the construction of the abstraction, the lasso found may be *spurious*, i.e., it may not correspond to a real execution of the program. In this case we use domain-specific procedures to detect spuriousness, and, if the lasso is found to be spurious, the abstraction is refined by eliminating it.

The framework described here needs to be instantiated for particular classes of programs, by providing the following ingredients:

- well-founded relations $\preceq_1, \ldots, \preceq_n$ on the working domain D. In principle, their choice is naturally determined by the working domain. As an example, if D is the set of terms (trees) over a finite ranked alphabet, then $\preceq_i$ can be classical well-founded orderings on terms (e.g., Recursive Path Ordering, Knuth-Bendix Ordering, etc.);
- a decision procedure for the problems $\Rightarrow \cap \preceq_i \stackrel{?}{=} \emptyset$, $1 \leq i \leq n$, where $\Rightarrow$ is any transition relation induced by a program statement. This is typically achieved by choosing suitable symbolic representations for relations on D, which are closed under intersection, and whose emptiness problem is decidable. For instance, this is the case when both $\Rightarrow$ and $\preceq_i$ can be encoded using finite (tree) automata;
- a decision procedure for the spuriousness problem: given a lasso $\sigma\lambda^\omega$, where σ and λ are finite sequences of program statements, does there exist an initial data

element $d_0 \in D$ such that the program, started with d_0, has an infinite execution along the path $\sigma\lambda^\omega$?

The main reason why we currently ask the user to provide the relations is that our technique is geared towards data domains which cannot be encoded by a finite number of descriptors, such as tree-structured domains, and more complex pointer structures. Well-founded relations for such classical domains (e.g., terms over a ranked alphabet) are provided in the literature. Moreover, we are not aware of efficient techniques for automatic discovery of well-founded relations on such domains, which is an interesting topic for future research.

Providing suitable representations for the well-founded relations, as well as for the program transitions enables the framework to compute an initial abstraction of the program. The initial abstraction is an automaton which has the same control states as the program, and each edge in the control flow graph of the program is covered by one or more transitions labeled with relational symbols.

The abstraction is next checked for the existence of potentially non-terminating executions. This check uses the information provided by the well-founded relations, and excludes all lassos for which there exists a strictly decreasing well-founded relation $\succ_i$, $1 \leq i \leq n$, that holds between the entry and exit of the loop body. This step amounts to checking non-emptiness of the intersection between the abstraction and a predefined Büchi automaton. If the intersection is empty, the original program terminates, otherwise a lasso-shaped counterexample of the form $\sigma\lambda^\omega$ is exhibited.

Deciding spuriousness of lassos is also a domain-dependent problem. For integer domains, techniques exist in cases where the transition relation of the loop is a difference bound matrix (DBM) [7] or an affine transformation with the finite monoid property [18]. For general affine transformations, the problem is currently open, as [10] mentions it[1]. For tree-structured data domains, the problem is decidable in cases, where the loop does not modify the structure of trees [20].

If a lasso is found to be spurious, the program model is refined by excluding the lasso from the abstraction automaton. In our framework based on Büchi Automata, this amounts to intersecting the abstraction automaton with the complement of the Büchi Automaton representing the lasso. Since a lasso is trivially a Weak Deterministic Büchi Automaton (WDBA), complementation increases the size of the automaton by at most one state, and is done in constant time. This refinement scheme can be extended to exclude entire families of spurious lassos, also described by WDBA.

We have instantiated the framework to the verification of programs handling tree data structures. The basic statements we consider are data assignments, non-destructive pointer assignments, creation of leaves, and deletion of leaves. As an extension, we also allow tree rotations. This is a sufficient fragment for verifying termination of many practical programs over tree-shaped data structures (e.g.,

[1] In fact the spuriousness problem for integer affine transition relations covers another open problem, that of detecting a zero in an integer linear recurrent sequence. The latter has been shown to be NP-hard in [2] but no decidability results have been found so far.

AVL trees or red-black trees) used, in general, for storage and a fast retrieval of data. Moreover, many programs working on singly- and doubly-linked lists fit into our framework as well. We provide two families of well-founded relations on trees, (i) a lexicographical ordering on positions of program variables and (ii) a subset relation on nodes labeled with a given data element (from a finite domain). Program statements as well as the well-founded relations are encoded using tree automata, which provide an efficient method for checking emptiness of intersections between relations. For programs on trees without destructive updates, the spuriousness problem has been shown decidable in [20].

The presented well-founded relations for trees fits very well for the encoding from [6], where data structures more complex than trees are encoded using a tree-like backbone and a regular set of extra edges. The encoding itself is quite general and allows us, on one hand, to handle structures like, e.g., trees with root pointers, or trees with linked leaves, and, on the other hand, to handle destructive updates.

A prototype tool has been implemented on top of the ARTMC [6, 4] invariant generator for programs with dynamic linked data structures. Experimental results include push-button termination proofs for the Deutsch-Schorr-Waite tree traversal, deleting nodes in red-black trees, as well as for the Linking Leaves procedures. Most of these programs could not be verified by existing approaches.

Contributions of the Paper:

1. The paper presents our original generic approach for termination analysis of programs. The approach is based on Büchi automata, well-founded relations, and CEGAR [11] loop.
2. We provide a set of well-founded relations on trees.
3. We instantiate the generic framework to programs manipulating tree like data structures as well as programs manipulating more general data structures (encoded as a tree-backbone and a regular set of extra edges).
4. We have implemented our technique in the prototype tool based on ARTMC [6, 4]. We can prove termination of examples which are, to the best of our knowledge, not handled by any other tool.

Related Work. Efficient techniques have been developed in the past for proving termination of programs with integer variables [26, 8, 9, 14, 28]. This remains probably the most extensively explored class of programs, concerning termination.

Recently, techniques for programs with singly-linked lists have been developed in [3, 17, 23]. These techniques rely on tracking numeric information related to the sizes of the list segments. An extension of this method to tackle programs handling trees has been given in our previous work [20]. Unlike the works on singly-linked lists from [3, 17], where refinement (of the counter model) is typically not needed, in [20] we considered a basic form of counterexample-driven refinement, based on a symbolic unfolding of the lasso-shaped counterexample.

Abstraction refinement for termination has been first considered in [14], where the refinement consists of discovering and adding new well-founded relations to the set of relations used by the analysis. Since techniques for the discovery of well-founded relations (based on, e.g., spurious program loops) are available only for integer domains, it is not clear for the time being whether the algorithm proposed in [14] can be also applied to programs handling pointer structures.

Several ideas in this paper can be also found elsewhere. Namely, (1) extracting variance assertions from loop invariants was reported in [1], (2) using Büchi automata to encode the non-termination condition of the program was introduced by [24], and (3) proving termination for programs handling tree-like data structures was also considered in [20]. However, the method presented here distinguishes itself from the body of existing work, on the following aspects:

1. The framework is general and can be instantiated to any class of programs, whose semantic domains are known to have well-founded orderings. In particular, we provide well-founded orderings on a domain of trees, e.g., lexicographical orderings, that cannot be directly encoded in quantifier-free linear arithmetic, whereas all variant assertions from [1] are confined to quantifier-free linear arithmetic. Moreover, we consider examples in which the variant relations considered track the evolution of an unbounded number of data elements (tree node labels), whereas in [1] only a finite number of seed variables could be considered.
2. We provide an automated method of abstracting programs into Büchi automata, whereas the size-change graphs from [24] are produced manually. Moreover, the use of Büchi automata to encode the termination condition provides us with a natural way of *refining* the abstract model, by intersection of the model with the complement of the counterexample, encoded by a Weak Deterministic Büchi Automaton.
3. We generalize the refinement based on Weak Deterministic Büchi Automata to exclude infinite sets of spurious counterexamples, all at once. On the other hand, the refinement given in [20] could only eliminate one counterexample at a time, at the cost of expanding the model by unfolding the lasso a number of times exponential in the number of program variables. In our setting, the size of the refined model is theoretically bounded by the product between the size of the model and the size of the lasso. Since in practice the lassos are found to be rather small, the blowup caused by the refinement does not appear to be critical.

Automated checking of termination of programs manipulating trees has been also considered in [25], where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate using a manually created progress monitor, encoded in first-order logic. In our approach, this example could be verified using the common well-founded relations on trees, that is without adding case-specific information to the framework.

The rest of the paper is organized as follows: Section 2 describes preliminaries. Section 3 contains general description of our termination analysis framework. Section 4 describes the way, how the framework is instantiated for trees and more complex data structures. Section 5 presents our experimental results, and finally we conclude the paper by Section 6.

2 PRELIMINARIES

2.1 Well-founded Relations

Definition 1. Given a set X, a relation $\preceq \subseteq X \times X$ is *well-founded* iff

1. $\preceq$ is a partial order on X
2. $\forall x \in X$ there is no infinite sequence $x_0, x_1, x_2, \ldots$ such that (i) $x_0 = x$ and (ii) $\forall i \geq 0 : x_{i+1} \preceq x_i \wedge x_{i+1} \neq x_i$.

As an example of a well-founded relation, we can take a standard $\leq$ ordering on natural numbers; but the same standard $\leq$ ordering on positive real numbers is not well-founded, because there exists, e.g., the sequence $1 > 0.1 > 0.01 > 0.001 > \ldots$

2.2 Büchi Automata

This section introduces the necessary notions related to the theory of Büchi automata [22]. Let $\Sigma = \{a, b, \ldots\}$ be a finite alphabet. We denote by Σ^* the set of finite words over Σ, and by Σ^ω the set of all infinite words over Σ. For an infinite word $w \in \Sigma^\omega$, let $\inf(w)$ be the set of symbols occurring infinitely often in w. If $u, v \in \Sigma^*$ are finite words, uv^ω denotes the infinite word $uvvv\ldots$

Definition 2. A *Büchi automaton* (BA) over Σ is a tuple $A = \langle S, I, \rightarrow, F \rangle$, where: S is a finite set of *states*, $I \subseteq S$ is a set of *initial states*, $\rightarrow \subseteq S \times \Sigma \times S$ is a *transition relation* – we denote $(s, a, s') \in \rightarrow$ by $s \xrightarrow{a} s'$, and $F \subseteq S$ is a set of *final states*. The *size* of the automaton A is denoted as $\|A\|$ and it is equal to the number of states – i.e., $\|A\| = \|S\|$.

A run of A over an infinite word $a_0 a_1 a_2 \ldots \in \Sigma^\omega$ is an infinite sequence of states $s_0 s_1 s_2 \ldots$ such that $s_0 \in I$ and for all $i \geq 0$ we have $s_i \xrightarrow{a_i} s_{i+1}$. A run π of A is said to be *accepting* iff $\inf(\pi) \cap F \neq \emptyset$. An infinite word w is *accepted* by a Büchi automaton A iff A has an accepting run on w. The *language* of A, denoted by $\mathcal{L}(A)$, is the set of all words accepted by A.

It is well-known that Büchi-recognizable languages are closed under union, intersection and complement. For two Büchi automata A and B, let $A \otimes B$ be the automaton recognizing the language $\mathcal{L}(A) \cap \mathcal{L}(B)$. It can be shown that $\|A \otimes B\| \leq 3 \cdot \|A\| \cdot \|B\|^2$.

[2] Intersection of A and B is done by a synchronous product of these two automata. For each pair of states, one has to remember an extra information whether a finite state has

A Büchi automaton $A = \langle S, I, \rightarrow, F\rangle$ is said to be *complete* if for every $s \in S$ and $a \in \Sigma$ there exists $s' \in S$ such that $s \xrightarrow{a} s'$. A is said to be *deterministic* (DBA) if I is a singleton, and for each $s \in S$ and $a \in \Sigma$, there exists at most one state $s' \in S$ such that $s \xrightarrow{a} s'$. A is moreover said to be *weak* if, for each strongly connected component $C \subseteq S$, either $C \subseteq F$ or $C \cap F = \emptyset$. It is well-known that complete weak deterministic Büchi automata can be complemented by simply reverting accepting and non-accepting states. Then, for any Weak Deterministic Büchi automaton (WDBA), we have that $\|\overline{A}\| \leq \|A\| + 1$, where $\overline{A}$ is the automaton accepting the language $\Sigma^\omega \setminus \mathcal{L}(A)$ – i.e., the *complement* of A.

2.3 Trees and Tree Automata

Definition 3 (Binary alphabet and tree)**.** For a partial mapping $f : A \to B$ we denote $f(x) = \perp$ the fact that f is undefined at some point $x \in A$. The domain of f is denoted $dom(f) = \{x \in A | f(x) \neq \perp\}$. For a set A we denote by $A_\perp$ the set $A \cup \{\perp\}$.

Given a finite set of *colors* $\mathcal{C}$, we define the *binary alphabet* $\Sigma_\mathcal{C} = \mathcal{C} \cup \{\Box\}$, where the *arity* function is $\forall c \in \mathcal{C}.\#(c) = 2$ and $\#(\Box) = 0$. Π denotes the set of tree positions $\{0, 1\}^*$. Let $\epsilon \in \Pi$ denote the empty sequence, and $p.q$ denote the concatenation of sequences $p, q \in \Pi$. $p \leq_{pre} q$ denotes the fact that p is a prefix of q and $p \leq_{lex} q$ is used to denote the fact that p is less than q in the lexicographical order. We denote by $p \simeq_{pre} q$ the fact that either $p \leq_{pre} q$, or $p \geq_{pre} q$. A *tree* t over $\mathcal{C}$ is a partial mapping $t : \Pi \to \Sigma_\mathcal{C}$ such that $dom(t)$ is a finite prefix-closed subset of Π, and for each $p \in dom(t)$:

- if $\#(t(p)) = 0$, then $t(p.0) = t(p.1) = \perp$,
- otherwise, if $\#(t(p)) = 2$, then $p.0, p.1 \in dom(t)$.

When writing $t(p) = \perp$, we mean that t is undefined at position p.

Let t_ϵ be the empty tree, $t_\epsilon(p) = \perp$ for all $p \in \Pi$. A *subtree* of t starting at position $p \in dom(t)$ is a tree $t_{|p}$ defined as $t_{|p}(q) = t(pq)$ if $pq \in dom(t)$, and undefined otherwise. $t[p \leftarrow c]$ denotes the tree that is labeled as t, except at position p where it is labeled with c. $t\{p \leftarrow u\}$ denotes the tree obtained from t by replacing the $t_{|p}$ subtree with u. We denote by $\mathcal{T}(\mathcal{C})$ the set of all trees over the binary alphabet $\Sigma_\mathcal{C}$.

Definition 4. A (binary) *tree automaton* [13, 27] over an alphabet $\Sigma_\mathcal{C}$ is a tuple $A = (Q, F, \Delta)$ where Q is a set of states, $F \subseteq Q$ is a set of final states, and Δ is a set of transition rules of the form:

1. $\Box \to q$ or
2. $c(q_1, q_2) \to q$, $c \in \mathcal{C}$.

been seen in the automaton A, in the automaton B or nowhere. Therefore the size of the resulting automaton is bounded by $3 \cdot \|A\| \cdot \|B\|$. See, e.g., [22] for details.

The *size* of the automaton A is denoted as $\|A\|$ and it is equal to the number of states – i.e., $\|A\| = \|Q\|$.

A *run* of A over a tree $t : \Pi \to \Sigma_{\mathcal{C}}$ is a mapping $\pi : dom(t) \to Q$ such that for each position $p \in dom(t)$, where $q = \pi(p)$, we have:

- if $\#(t(p)) = 0$ (i.e., if $t(p) = \square$), then $\square \to q \in \Delta$,
- otherwise, if $\#(t(p)) = 2$ and $q_i = \pi(p.i)$ for $i \in \{0, 1\}$, then $t(p)(q_0, q_1) \to q \in \Delta$.

A run π is said to be *accepting* if and only if $\pi(\epsilon) \in F$. The *language* of A, denoted as $\mathcal{L}(A)$, is the set of all trees over which A has an accepting run. A set of trees $T \subseteq \mathcal{T}(\mathcal{C})$ (a tree relation $R \subseteq \mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2)$) is said to be *rational* if there exists a tree automaton A such that $\mathcal{L}(A) = T$ (respectively, $\mathcal{L}(A) = R$).

Example: Let $\mathcal{C} = \{\circ, \bullet\}$ and the following tree automaton over $\Sigma_{\mathcal{C}}$: $A = (\{q, r\}, \{r\}, \Delta_A)$ with $\Delta_A = \{\square \to q, \circ(q, q) \to q, \bullet(q, q) \to r, \circ(r, q) \to r, \circ(q, r) \to r\}$. This automaton accepts all binary trees which contain exactly one node labeled by $\bullet$. An example of runs can be seen in Figure 1. Note that for some trees there is no run which maps a state to the most-top node (symbol X corresponds to non-existing mapping), and hence such a tree is not accepted.

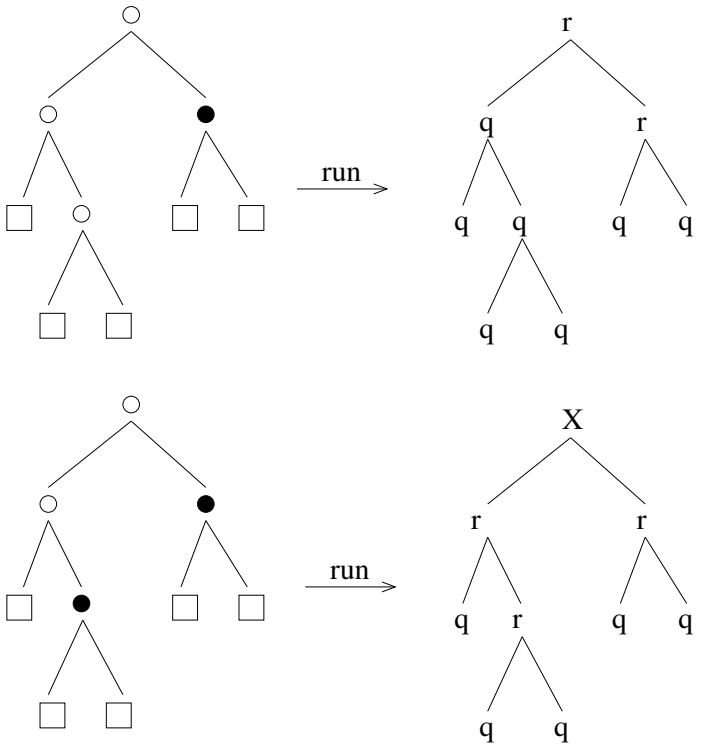


Figure 1.

Remark: In this paper, we use only tree automata restricted to binary trees; but in general, the tree automata can accept trees of arbitrary arity (see [13, 27] for general definitions). Note that standard finite automata are special case of tree automata, where all rules are unary (except the initial one $\Box \to q$, which represents the initial state in standard finite automaton).

2.3.1 Rational Tree Relations

A pair of trees $(t_1, t_2) \in \mathcal{T}(\mathcal{C}_1) \times \mathcal{T}(\mathcal{C}_2)$ can be encoded by a tree over the alphabet $(\mathcal{C}_1 \cup \{\Box, \bot\}) \times (\mathcal{C}_2 \cup \{\Box, \bot\})$, where $\#(\langle \bot, \bot \rangle) = 0$, $\#(\langle \alpha, \bot \rangle) = \#(\langle \bot, \alpha \rangle) = \#(\alpha)$ if $\alpha \neq \bot$, and $\#(\langle \alpha_1, \alpha_2 \rangle) = 2$ if $\alpha_1 \in \mathcal{C}_1 \ \wedge \ \alpha_2 \in \mathcal{C}_2$. The projection functions are defined as usual, i.e., for all $p \in dom(t)$ we have $pr_1(t)(p) = c_1$ if $t(p) = \langle c_1, c_2 \rangle$ and $pr_2(t)(p) = c_2$ if $t(p) = \langle c_1, c_2 \rangle$. Finally, let $\mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2) = \{t | pr_1(t) \in \mathcal{T}(\mathcal{C}_1)$ and $pr_2(t) \in \mathcal{T}(\mathcal{C}_2)\}$ be a set of all pair trees. A set $R \subseteq \mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2)$ is further called a *tree relation*.

Definition 5. A tree relation $R \subseteq \mathcal{T}(\mathcal{C}_1 \times \mathcal{C}_2)$ is said to be *rational* if there exists a tree automaton A such that $\mathcal{L}(A) = R$.

Definition 6. For two relations $R' \subseteq \mathcal{T}(\mathcal{C} \times \mathcal{C}')$ and $R'' \subseteq \mathcal{T}(\mathcal{C}' \times \mathcal{C}'')$ we define the composition $R' \circ R'' = \{\langle pr_1(t'), pr_2(t'') \rangle | t' \in R', t'' \in R'', pr_2(t') = pr_1(t'')\}$.

It is well known that rational tree languages are closed under union, intersection, complement and projection. As a consequence, rational tree relations are closed under composition.

3 THE TERMINATION ANALYSIS FRAMEWORK

In this section, we describe our termination analysis framework and demonstrate its principles on a simple running example.

3.1 Programs and Abstractions

First we introduce a model for programs handling data from a possibly infinite domain D equipped by a set of partial orders $\preceq_1, \ldots, \preceq_n$, where $\preceq_i \subseteq D \times D$ for $1 \leq i \leq n$. In the following, we use the notion $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ to denote the data domain with the partial orders. Then we define program abstractions as Büchi automata.

Definition 7 (Instruction)**.** Let $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ be a data domain. An instruction is a pair $\langle g, a \rangle$ where $g \subseteq D$ is called the *guard* and $a : D \to D$ is called the *action*.

The guard represents a condition, which must be true before the action is executed. The guards are used to model conditional statements from programming languages – e.g., one instruction $\langle g, a_1 \rangle$ is used for *then* branch and another one $\langle D \backslash g, a_2 \rangle$ for *else* branch. An unspecified guard is assumed to be the entire domain D.

Definition 8 (Program)**.** Let $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ be a data domain and $\mathfrak{I}$ be a set of instructions. A program over the set of instructions $\mathfrak{I}$ is a graph $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$, where L is the set of *control locations*, $l_0 \in L$ is the *initial location*, and $\Rightarrow \subseteq L \times \mathfrak{I} \times L$ is the *edge relation* denoted as $l \overset{g:a}{\Longrightarrow} l'$. We assume furthermore that there is at most one instruction in between any two control locations, i.e., if $l \overset{g_1:a_1}{\Longrightarrow} l'$ and $l \overset{g_2:a_2}{\Longrightarrow} l'$ then $g_1 = g_2$ and $a_1 = a_2$. This condition is common in programming languages.

In the following, we will use both textual and graphical representations of programs.

Definition 9 (Configuration, execution, reachable configuration)**.** Let $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ be a data domain, $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ a program and $D_0 \subseteq D$ be a set of initial data values. A *program configuration* is a pair $\langle l, d \rangle \in L \times D$, where l is a control location and d is a data value.

An *execution* is a (possibly infinite) sequence of program configurations $\langle l_0, d_0 \rangle, \langle l_1, d_1 \rangle, \langle l_2, d_2 \rangle, \ldots$ starting with the initial program location l_0 and some configuration $d_0 \in D$ such that for all $i \geq 0$ there exists an edge $l_i \overset{g:a}{\Longrightarrow} l_{i+1}$ in the program, such that $d_i \in g$ and $d_{i+1} = a(d_i)$.

A configuration $\langle l, d \rangle$ is said to be *reachable* if there exists $d_0 \in D_0$, and the program P has an execution from $\langle l_0, d_0 \rangle$ to $\langle l, d \rangle$.

Definition 10 (Invariant)**.** Let $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ be a data domain, $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ a program and $D_0 \subseteq D$ be a set of initial data values. An *invariant* of the program (with respect to the set D_0) is a function $\iota : L \to 2^D$ such that for each $l \in L$, if $\langle l, d \rangle$ is reachable, then $d \in \iota(l)$. If the dual implication holds, we say that ι is an *exact invariant*.

Definition 11. Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ working over a domain $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ we define the alphabet $\Sigma_{(P,D)} = L \times \{>, \bowtie, =\}^n$. For a tuple $\rho \in \{>, \bowtie, =\}^n$, we define $[\rho] \in D \times D$ as : $d[\rho]d'$ if and only if, $\rho = \langle r_1, \ldots, r_n \rangle$ and for all $1 \leq i \leq n$:

- $d \succ_i d'$ iff r_i is $>$,
- $d \not\preceq_i d'$ iff r_i is $\bowtie$,
- $d \approx_i d'$ iff r_i is $=$.

Definition 12 (Abstraction)**.** Let $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ be a program, and $\langle D, \preceq_1, \ldots, \preceq_n \rangle$ be a domain. A Büchi automaton $A = \langle S, I, \rightarrow, F \rangle$ over $\Sigma_{(P,D)}$ is said to be an abstraction of P if and only if for every infinite execution of P: $\langle l_0, d_0 \rangle$ $\langle l_1, d_1 \rangle \langle l_2, d_2 \rangle \ldots$, there exists an infinite word $\langle l_0, \rho_0 \rangle \langle l_1, \rho_1 \rangle \langle l_2, \rho_2 \rangle \ldots \in \mathcal{L}(A)$ such that $d_i[\rho_i]d_{i+1}$,for all $i \geq 0$.

Consequently, if P has a non-terminating execution, then its abstraction A will be non-empty. However, for reasons related to the complexity of the universal termination problem, one cannot in general build an abstraction of a program that will be empty if and only if the program terminates.

3.1.1 The Running Example: A Program and Its Abstraction

We will now demonstrate the abstraction of a real program on our running example. Let us consider the program in Figure 2, working on a binary tree data structure, in which each node has two pointers to its `left`- and `right`-sons and one pointer `up` to its parent. We assume that leaves have null `left` and `right` pointers, and the `root` has a null `up` pointer.

```
1  x := root;
2  while (x.left != null) and (x.left.right != null)
3          x := x.left.right;
4  while (x != null)
5          x := x.up;
```

Figure 2.

The first loop (lines 2,3) terminates because the variable x is bound to reach a node with `x.left = null` (or `x.left.right = null`), since the tree is finite and no new nodes are created. The second loop (lines 4,5) terminates because no matter where x points to in the beginning, by going up, it will eventually reach the `root` and then become null. Figure 3 represents the c-like program from Figure 2 as a program according to Definition 8.

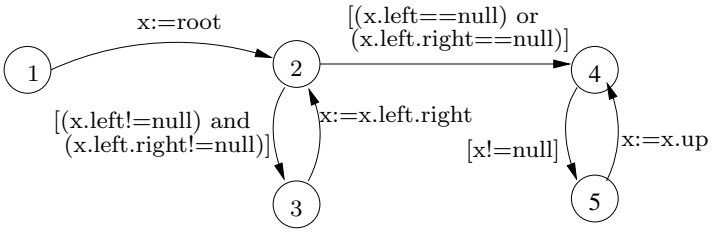


Figure 3. Guards are represented inside "[ ]" brackets, and actions without brackets. Unspecified action is equal to identity relation.

Now, we are going to create an abstraction of program in Figure 3 Let us suppose the following well-founded ordering: for any two trees t_1 and t_2, we have $t_1 \geq_x t_2$ if and only if the position of the pointer variable x in t_2 is a prefix of the position of the variable x in t_1. Using the $\geq_x$ ordering, we build an abstraction of the program given in Figure 4. The states in the abstract program correspond to line numbers in the original program, and every state is considered to be accepting (w.r.t Büchi accepting condition), initially.

Note that the action $x := root$ is abstracted by two edges – first, labeled by $=_x$, describes the case when x was originally placed in the root node and the other one, labeled by $>_x$, describes the cases when x was originally deeper in the tree.

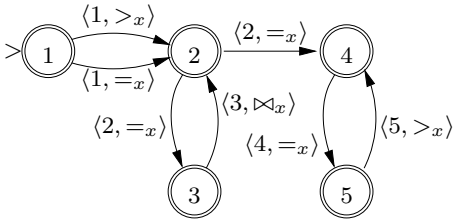


Figure 4.

3.2 Building Abstractions Automatically

The first question is how to build abstractions of programs effectively. Here, we propose a method that performs well under the following assumptions: There exists a symbolic representation $\mathcal{S}$ (e.g., some logic or automata) such that

1. the program instructions can be represented using the $\mathcal{S}$,
2. the well-founded relations on the working domain can be also represented using $\mathcal{S}$,
3. $\mathcal{S}$ is closed under projection, intersection and complement, and
4. the emptiness problem for $\mathcal{S}$ is decidable.

In the concrete case of programs with trees (see Section 4), we use finite tree automata [13] as a suitable symbolic representation $\mathcal{S}$.

Definition 13 (Initial abstraction)**.** Given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ working over the domain $\langle D, \preceq_1, \ldots, \preceq_n \rangle$, and an invariant $\iota : L \to 2^D$, with respect to a set of initial data values D_0, the *initial abstraction* is the Büchi automaton $A^{\iota}_P = \langle L, \{l_0\}, \rightarrow, L \rangle$, where, for all $l, l' \in L$ and $\rho \in \{>, \bowtie, =\}^n$, we have:

$$l \xrightarrow{\langle l, \rho \rangle} l' \iff l \overset{g:a}{\Longrightarrow} l' \text{ and } pr_1(R_{\langle g,a \rangle} \cap [\rho]) \cap \iota(l) \neq \emptyset \tag{1}$$

where $R_{\langle g,a \rangle} = \{(d, d') \in D | d \in g, d' = a(d)\}$ and, for a relation $R \subseteq D \times D$, we denote by $pr_1(R) = \{x | \exists y \in D. \langle x, y \rangle \in R\}$.

Intuitively, a transition between l and l' is labeled with a tuple of relational symbols ρ if and only if there exists a program instruction between l and l' and a pair of reachable configurations $\langle l, d \rangle, \langle l', d' \rangle \in L \times D$ such that $d[\rho]d'$ and the program can move from $\langle l, d \rangle$ to $\langle l', d' \rangle$ by executing the instruction $\langle g, a \rangle$. The intuition is that every transition relation induced by the program is "covered" by all partial orderings that have a non-empty intersection with it. Notice also that, for any $d, d' \in D$, there exists $\rho \in \{>, \bowtie, =\}^n$ such that $d[\rho]d'$. For reasons related to abstraction refinement, that will be made clear in the following, the transition

in the Büchi automaton A^{ι}_P is also labeled with the source program location l[3]. As an example, Figure 2 b) gives the initial abstraction for the program in Figure 2 a).

The program invariant $\iota(l)$ from (1) is needed in order to limit the coverage only to the relations involving configurations reachable at line l. In principle, we can compute a very coarse initial abstraction by considering that $\iota(l) = D$ at each program line. However, using stronger invariants enables us to compute more precise program abstractions. The following lemma proves that the initial abstraction respects Definition 12.

Lemma 1. Let P be a program working over the domain $\langle D, \preceq_1, \ldots, \preceq_n\rangle$, and $D_0 \subseteq D$ be an initial set, and $\iota : L \to 2^D$ be an invariant with respect to the initial set D_0, the Büchi automaton A^{ι}_P is an abstraction of P.

Proof. Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \ldots$ be an arbitrary infinite execution of the program P. Then for each $i \geq 0$: $l_i \overset{g_i:a_i}{\Longrightarrow} l_{i+1}$, $d_i \in g_i$, and $a_i(d_i) = d_{i+1}$ – i.e., $(d_i, d_{i+1}) \in R_{\langle g_i, a_i\rangle}$, where $R_{\langle g,a\rangle} = \{(d, d') \in D | d \in g, d' = a(d)\}$. Moreover, $d_i \in \iota(l_i)$ and $d_{i+1} \in \iota(l_{i+1})$. We know that there exists ρ_i such that $d_i[\rho_i]d_{i+1}$. Therefore for each $i \geq 0$ there exists an edge $l_i \xrightarrow{\langle l_i, \rho_i\rangle} l_{i+1}$ in the automaton A^{ι}_P, i.e., the word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in \mathcal{L}(A^{\iota}_P)$, where $d_i[\rho_i]d_{i+1}$. □

3.3 Checking Termination on Program Abstractions

In light of Definition 12, if a Büchi automaton A is an abstraction of a program P, then each accepting run of A reveals a *potentially* infinite execution of P. However, the set of accepting runs of a Büchi automaton is infinite in general, therefore an effective termination analysis cannot attempt to check whether each run of A corresponds to a real computation of P. We propose an effective technique, based on the following assumption:

Assumption 1. The given domain is $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ for a fixed $n > 0$, and the partial orders $\preceq_i$ are well-founded, for all $i = 1, \ldots, n$.

Consequently, any infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in \mathcal{L}(A)$ from which we can extract[4] a sequence $(\rho_0)_i(\rho_1)_i(\rho_2)_i \ldots \in (=^*>)^{\omega}$, for some $1 \leq i \leq n$, cannot correspond to a real execution of the program, in the sense of Definition 12. Therefore, we must consider only the words for which, for all $1 \leq i \leq n$, either:

[3] After the abstraction refinement, the relation between program locations and Büchi automata states is no more $1 : 1$ (as in the initial abstraction) but it is $m : n$. The labels are then used to relate the edges in the abstract model with the corresponding program actions.

[4] Each element $\langle l, \rho\rangle$ in the sequence is first converted into ρ by omitting the l-part. Then all the relations except the i^{th} one are projected out of ρ. The result is denoted as $(\rho)_i$

1. there exists $K \in \mathbb{N}$ such that, $(\rho_k)_i$ is $=$, for all $k \geq K$, or
2. for infinitely many $k \in \mathbb{N}$, $(\rho_k)_i$ is $\bowtie$.

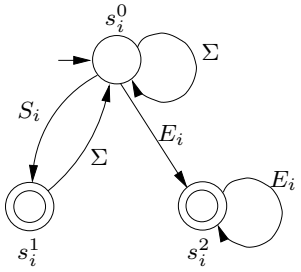


Figure 5.

The condition above can be encoded by a Büchi automaton defined as follows. Consider that $\Sigma_{(P,D)} = L \times \{>, \bowtie, =\}^n$ is fixed. Let $S_i = \{\langle l, (r_1, \ldots, r_n)\rangle \in \Sigma_{(P,D)} | r_i \text{ be } \bowtie\}$ and $E_i = \{\langle l, (r_1, \ldots, r_n)\rangle \in \Sigma_{(P,D)} | r_i \text{ be } =\}$, for $1 \leq i \leq n$. With this notation, let B_i be the Büchi automaton recognizing the ω-regular language $\Sigma^*(S_i\Sigma^*)^\omega \cup \Sigma^* E_i^\omega$. This automaton is depicted in Figure 5. Since the above condition holds for all $1 \leq i \leq n$, we need to compute $B = \bigotimes_{i=1}^n B_i$. Finally, the automaton which accepts all words witnessing potentially non-terminating runs of the original program is $A \otimes B$.

If A is an abstraction of P and $\mathcal{L}(A \otimes B) = \mathcal{L}(A) \cap \mathcal{L}(B) = \emptyset$, we can infer that P has no infinite runs. Otherwise, it is possible to exhibit a lasso-shaped non-termination witness of the form $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$, where $\sigma, \lambda \in \Sigma^*$ are finite words labeling finite paths in $A \otimes B$. In the rest of the paper, we refer to σ as to the *stem* and to λ as to the *loop* of the lasso. The following lemma proves the existence of lasso-shaped counterexamples.

Lemma 2. Let $\langle D, \preceq_1, \ldots, \preceq_n\rangle$ be a well-founded domain, A be a Büchi automaton representing the abstraction of a program and $B = \bigotimes_{i=1}^n B_i$ be a Büchi automaton representing the non-termination property such that $\mathcal{L}(A \otimes B) \neq \emptyset$. Then there exists $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$ for some $\sigma, \lambda \in \Sigma^*_{(P,D)}$, such that $|\sigma|, |\lambda| \leq \|A\| \cdot (n+1) \cdot 2^n$.

Proof. If $\mathcal{L}(A \otimes B) \neq \emptyset$, then $A \otimes B$ has a run π in which at least one final state s occurs infinitely often. Let σ be the word labeling the prefix of π from the beginning to the first occurrence of s, and λ be the word labeling the segment between the first and second occurrences of s on π. Then $\sigma\lambda^\omega \in \mathcal{L}(A \otimes B)$.

To prove the bound on $|\lambda|$, we consider that B is the result of a generalized product $\bigotimes_{i=1}^n B_i$, whose states are of the form $\langle s_1, \ldots, s_n, k\rangle$ where $s_i \in \{s_i^0, s_i^1, s_i^2\}$ is a state of B_i (cf. Figure 5) and $k \in \{0, 1, \ldots, n\}$. Since λ is the label of a cycle in $A \otimes B$, the projection of the initial and final states on $B_1, \ldots, B_n$ must be the same. Then for each state in the cycle, either $s_i \in \{s_i^0, s_i^1\}$ or $s_i = s_i^2$, for each $1 \leq i \leq n$. This is because the projection of a cycle from $A \otimes B$ on $B_i, 1 \leq i \leq n$

is again a cycle, and the only cycles in B_i are composed either of $\{s_i^0, s_i^1\}$ or $\{s_i^2\}$. Hence $|\lambda| \leq \|A\| \cdot (n+1) \cdot 2^n$. A similar reasoning is used to establish the bound on $|\sigma|$. □

Despite the exponential bound on the size of the counterexamples, in practice it is possible to use efficient algorithms for finding lassos in Büchi automata on-the-fly, such as for instance the Nested Depth First Search algorithm [16].

3.3.1 The Running Example: The Termination Check

Checking (non-)termination of the abstract program in Figure 4 is done by checking emptiness of the intersection between the abstraction and the complement of the Büchi automaton recognizing the language $(\langle \_, =_x \rangle^* \langle \_, >_x \rangle)^\omega$ (cf. Figure 6). In the case of our running example, the intersection is not empty, counterexamples being $\langle 1, >_x \rangle (\langle 2, =_x \rangle \langle 3, \bowtie_x \rangle)^\omega$ and $\langle 1, =_x \rangle (\langle 2, =_x \rangle \langle 3, \bowtie_x \rangle)^\omega$, which both correspond to the infinite execution of the first loop, i.e., lines $1(23)^\omega$.

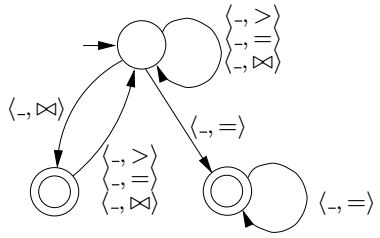


Figure 6.

3.4 Counterexample-Based Abstraction Refinement

If a Büchi automaton A is an abstraction of a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ (cf. Definition 12), $D_0 \in D$ is a set of initial values, and $\sigma\lambda^\omega \in \mathcal{L}(A)$ is a lasso, where $\sigma = \langle l_0, \rho_0 \rangle \dots \langle l_{|\sigma|-1}, \rho_{|\sigma|-1} \rangle$ and $\lambda = \langle l_{|\sigma|}, \rho_{|\sigma|} \rangle \dots \langle l_{|\sigma|+|\lambda|-1}, \rho_{|\sigma|+|\lambda|-1} \rangle$, the *spuriousness problem* asks whether P has an execution along the infinite path $(l_0 \dots l_{|\sigma|-1})(l_{|\sigma|} \dots l_{|\sigma|+|\lambda|-1})^\omega$ starting with some value $d_0 \in D_0$. Notice that each pair of control locations corresponds to exactly one program instruction, therefore the sequence of instructions corresponding to the infinite unfolding of the lasso is uniquely identified by the sequences of locations $l_0, \dots, l_{|\sigma|-1}$ and $l_{|\sigma|}, \dots, l_{|\sigma|+|\lambda|-1}$.

Algorithms for solving the spuriousness problem exist, depending on the structure of the domain D and on the semantics of the program instructions. Details regarding spuriousness problems for integer and tree-manipulating lassos can be found in [20].

Given a lasso $\sigma\lambda^\omega \in \mathcal{L}(A)$, the refinement builds another abstraction A' of P such that $\sigma\lambda^\omega \notin \mathcal{L}(A')$. Having established that the program path $(l_0 \dots l_{|\sigma|-1})(l_{|\sigma|} \dots l_{|\sigma|+|\lambda|-1})^\omega$, corresponding to $\sigma\lambda^\omega$, cannot be executed for any value from the initial

set allows us to refine by excluding potentially more spurious witnesses than just $\sigma\lambda^\omega$. Let C be the Büchi automaton recognizing the language $L_\sigma L_\lambda^\omega$, where:

$$\begin{aligned} L_\sigma &= \{\langle l_0, \rho_0\rangle \dots \langle l_{|\sigma|-1}, \rho_{|\sigma|-1}\rangle | \rho_i \in \{>, \bowtie, =\}^n, 0 \leq i < |\sigma|\} \\ L_\lambda &= \{\langle l_{|\sigma|}, \rho_0\rangle \dots \langle l_{|\sigma|+|\lambda|-1}, \rho_{|\lambda|-1}\rangle | \rho_i \in \{>, \bowtie, =\}^n, 0 \leq i < |\lambda|\}. \end{aligned}$$

Then $A' = A \otimes \overline{C}$, where $\overline{C}$ is the complement of C, is the refinement of A that excludes the lasso $\sigma\lambda^\omega$, and all other lassos corresponding to the program path $(l_0 \dots l_{|\sigma|-1})(l_{|\sigma|} \dots l_{|\sigma|+|\lambda|-1})^\omega$.

On the down side, complementation of Büchi automata is, in general, a costly operation: the size of the complement is bounded by $2^{\mathcal{O}(n \log n)}$, where n is the size of the automaton [29]. However, the particular structure of the automata considered here comes to rescue. It can be seen that $L_\sigma L_\lambda^\omega$ can be recognized by a WDBA, hence complementation is done in constant time, and $\|A'\| \leq 3 \cdot (|\sigma| + |\lambda| + 1) \cdot \|A\|$.

Lemma 3. Let A be a Büchi automaton that is an abstraction of a program P, and $\sigma\lambda^\omega \in \mathcal{L}(A)$ be a spurious counterexample. Then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus L_\sigma \cdot L_\lambda^\omega$ is an abstraction of P.

Proof. Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \dots$ be an infinite run of P. Since A is an abstraction of P, by Definition 12 there exists an infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \dots \in \mathcal{L}(A)$ such that $d_i[\rho_i]d_{i+1}$, for all $i \geq 0$. Since $\sigma\lambda^\omega$ is a spurious lasso, then the sequence of control locations $l_0, l_1, l_2, \dots$ cannot correspond to the sequence of first positions from $\sigma\lambda^\omega$. Hence $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \dots \notin L_\sigma \cdot L_\lambda^\omega$. Since the infinite run $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \dots$ was chosen arbitrarily, it follows that the automaton recognizing $\mathcal{L}(A) \setminus L_\sigma \cdot L_\lambda^\omega$ is an abstraction of P. □

3.4.1 The Running Example: The Refinement

In the case of our running example, we discovered the lasso $1(23)^\omega$. This execution is found to be spurious by a specialized procedure that checks whether a given program lasso can be fired infinitely often. For this purpose, the method given in [20] could be used here, since the loop $1(23)^\omega$ does not change the structure of the tree. The refinement of the abstraction consists in eliminating the infinite path $1(23)^\omega$ from the model. This is done by intersecting the model with the automaton that recognizes the *complement* of the language $\{\langle 1, >_x\rangle, \langle 1, =_x\rangle\}(\langle 2, =_x\rangle\langle 3, \bowtie_x\rangle)^\omega$, which corresponds to the program path $1(23)^\omega$. The result of this intersection is shown Figure 7. Notice that, in this case, the refinement does not increase the size of the abstraction. Since now, only 4 and 5 are accepting states, another intersection with the automaton in Figure 6 will be empty and hence the refined abstraction does not have further non-terminating executions, proving thus termination of the original program.

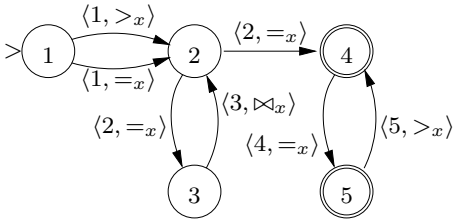


Figure 7.

3.5 Refinement Heuristics

The refinement technique, based on the closure of ω-regular languages under intersection and complement can be generalized to exclude an entire family of counterexamples, described as an ω-regular language, all at once. In the following we provide such a refinement heuristics.

The main difficulty here is to generalize from lasso-shaped counterexamples to more complex sets of counterexamples. In the following, we provide two refinement heuristics that eliminate entire families of counterexamples, at once. We assume in the following that we are given an invariant $\iota : L \to 2^D$ of the program.

Infeasible Elementary Loop Refinement. Let $\sigma\lambda^\omega$ be a lasso representing a spurious counterexample. To apply this heuristic method, we suppose that there exists an upper bound $B \in \mathbb{N}, B > 0$, on the number of times λ can be iterated, starting with any data value from $\iota(l_{|\sigma|})$, where $\iota(l_{|\sigma|})$ denotes the program invariant on location reachable in the program by the sequence of instructions σ from the initial location. The existence of such a bound can be discovered by, e.g., a symbolic execution of the loop. In case such a bound B exists, let C be a WDBA such that $\mathcal{L}(C) = \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$. Then the Büchi automaton $A \otimes \overline{C}$ is an abstraction of P, which excludes the spurious trace $\sigma\lambda^\omega$, as shown by the following lemma:

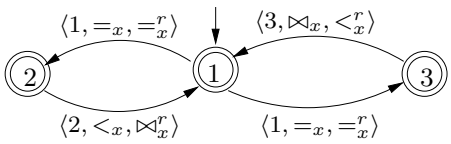
Lemma 4. Let $P = \langle \mathfrak{I}, L, l_0, \Rightarrow \rangle$ be a program, $\iota : L \to 2^D$ be an invariant of P, A be an abstraction of P, and $\lambda \in \Sigma^*_{(P,D)}$ be a lasso starting and ending with $\ell \in L$. If there exists $B > 0$ such that λ^B is infeasible, for any $d \in \iota(\ell)$, then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$ is an abstraction of P.

Proof. Let $\langle l_0, d_0\rangle\langle l_1, d_1\rangle\langle l_2, d_2\rangle \ldots$ be an infinite run of P. Since A is an abstraction of P, by Definition 12 there exists an infinite word $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \in \mathcal{L}(A)$ such that $d_i[\rho_i]d_{i+1}$, for all $i \geq 0$. Since λ^B cannot be fired for any data element $d \in \iota(\ell)$, then the infinite sequence $l_0, l_1, l_2, \ldots$ may not contain the subsequence $(l_{|\sigma|}l_{|\sigma|+1} \ldots l_{|\sigma|+|\lambda|-1})^B$, corresponding to the first positions from λ^B. Consequently $\langle l_0, \rho_0\rangle\langle l_1, \rho_1\rangle\langle l_2, \rho_2\rangle \ldots \notin \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$. Since the infinite run $\langle l_0, d_0\rangle\langle l_1, d_1\rangle$ $\langle l_2, d_2\rangle \ldots$ was chosen arbitrarily, it follows that the automaton recognizing $\mathcal{L}(A) \setminus \Sigma^*_{(P,D)} \cdot L^B_\lambda \cdot \Sigma^\omega_{(P,D)}$ is an abstraction of P. □

As an example, let us consider the following program:

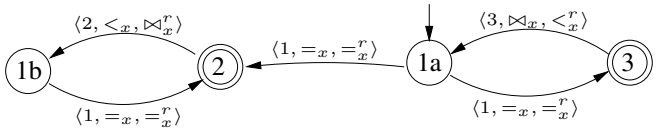
```
1  while (x != null)
          if (root.data)
2           x := x.left;
3         else x := x.up;
```

Here we assume that the root of the tree has a Boolean `data` field, which is used by the loop to determine the direction (i.e., `left`, `up`) of the variable x. The initial abstraction for this program is:



This abstraction uses two well-founded relations, $\leq_x$ defined in Section 3, and $\leq_x^r$ which is a stronger version of the reversed relation ($\geq_x$): $t_1 \leq_x^r t_2$ iff $dom(t_1) = dom(t_2)$[5] and $t_2 \leq_x t_1$.

Then a spurious lasso is $\sigma\lambda^\omega$, where σ is the empty word, and λ is $\langle 1, =_x, =_x^r\rangle\langle 2, <_x, \bowtie_x^r\rangle\langle 1, =_x, =_x^r\rangle\langle 3, \bowtie_x, <_x^r\rangle$, which corresponds to the program path 1213. This path is infeasible for any tree and any position of x, therefore we refine the abstraction by eliminating the language $\Sigma^*_{(P,D)} L_\lambda \Sigma^\omega_{(P,D)}$. The refined automaton is given below.



The intersection of the refined automaton with the automaton $B = B_1 \otimes B_2$ is empty, where $B_i, i = 1, 2$ are the automata from Figure 5; hence, by Definition 12, we can conclude that the program terminates.

This heuristic was used to prove termination of the *Red-black delete* algorithm, reported in Section 5. Interestingly, this algorithm could not be proved to terminate using standard refinement (cf. Lemma 3).

Infeasible Nested Loops Refinement. Let us assume that the location $l_{|\sigma|}$ is the source (and destination) of $k > 1$ different elementary loops in A: $\lambda_1, \ldots, \lambda_k$. Moreover, let us assume that:

1. these loops can only be fired in a given total order, denoted $\lambda_{i_1} \rhd \lambda_{i_2} \rhd \ldots \lambda_{i_k}$, for each input value in the set $D_\sigma = \{d | \langle l_0, d_0\rangle \stackrel{\sigma}{\Rightarrow} \langle l_{|\sigma|}, d\rangle, d_0 \in D_0\}$[6];

[5] For a tree t, $dom(t)$ is the set of positions in the tree.

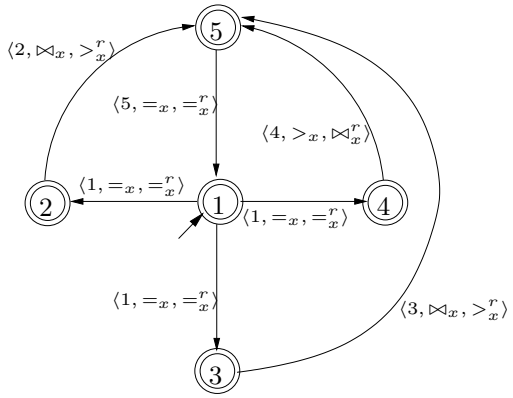
[6] Notice that checking the existence of such an ordering amounts to performing at most k^2 feasibility checks, for all paths of the form $\lambda_i \cdot \lambda_j, 1 \leq i, j \leq k, i \neq j$.

2. the infinite word $\sigma(\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is a spurious counterexample for non-termination, i.e., the corresponding program path is infeasible for any initial value from D_0.

Under these assumptions, let C be the Büchi automaton recognizing the language $L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$. The following lemma shows that $A \otimes \overline{C}$ is an abstraction of the program, that excludes the spurious lasso $\sigma\lambda^\omega$.

```
1  while (x != null)
       if (root.data == 0)
2        x := x.left;
       else if (root.data == 1)
3        x := x.right;
       else if (root.data == 2)
4        x := x.up;
5      root.data := (root.data+1) % 3;
```

a)



b)

Figure 8.

Lemma 5. Let $P = \langle \mathfrak{J}, L, l_0, \Rightarrow \rangle$ be a program, $\iota : L \to 2^D$ be an invariant of P, A be an abstraction of P, and $\lambda_1, \ldots, \lambda_k \in \Sigma^*_{(P,D)}$ be words labeling different cycles of A starting and ending with the same location $\ell \in L$. Moreover, let $\sigma \in \Sigma^*_{(P,D)}$ be the label of a path from l_0 to ℓ in A. If there exists a total order $\lambda_{i_1} \rhd \ldots \rhd \lambda_{i_k}$ in which the cycles can be executed, for any $d \in \iota(\ell)$, and $\sigma \cdot (\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is moreover spurious, then the Büchi automaton recognizing the language $\mathcal{L}(A) \setminus L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$ is an abstraction of P.

Proof. Let $\langle l_0, d_0 \rangle \langle l_1, d_1 \rangle \langle l_2, d_2 \rangle \ldots$ be an infinite run of P. Since A is an abstraction of P, by Definition 12 there exists an infinite word $\langle l_0, \rho_0 \rangle \langle l_1, \rho_1 \rangle \langle l_2, \rho_2 \rangle \ldots \in \mathcal{L}(A)$ such that $d_i[\rho_i]d_{i+1}$, for all $i \geq 0$. We show that $\langle l_0, \rho_0 \rangle \langle l_1, \rho_1 \rangle \langle l_2, \rho_2 \rangle \ldots \notin L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$. Assume by contradiction that $\langle l_0, \rho_0 \rangle \langle l_1, \rho_1 \rangle \langle l_2, \rho_2 \rangle \ldots \in L_\sigma \cdot (L_{\lambda_1} \cup \ldots \cup L_{\lambda_k})^\omega$. Then the infinite sequence $l_0 l_1 l_2 \ldots$ of first positions can be decomposed into a prefix corresponding to σ, followed by an infinite succession of blocks corresponding to some $\lambda_i, 1 \leq i \leq k$. Since $\lambda_{i_1} \rhd \lambda_{i_2} \rhd \ldots \lambda_{i_k}$ is the only possible execution order, these blocks must respect the order $\rhd$. However, this is in contradiction with the fact that $\sigma(\lambda_{i_1} \cdot \ldots \cdot \lambda_{i_k})^\omega$ is a spurious lasso. The conclusion follows. □

As an example, let us consider the program in Figure 8 a). Its initial abstraction, using the relations $\leq_x$ and $\leq^r_x$, is given in Figure 8 b). The three nested loops are in the unique execution order: $1251 \rhd 1351 \rhd 1451 \rhd 1251$. Moreover, the composed loop $(1251)(1351)(1451)$ cannot be iterated infinitely often because the pointer variable x,

moving twice down and once up in the tree, is bound to become null. This fact can be also detected automatically using, e.g., the technique from [20]. Therefore we can refine the initial abstraction by eliminating the language $\{\langle 1, =_x, =_x^r\rangle\langle 2, \bowtie_x, >_x^r\rangle\langle 5, =_x, =_x^r\rangle, \langle 1, =_x, =_x^r\rangle\langle 3, \bowtie_x, >_x^r\rangle\langle 5, =_x, =_x^r\rangle, \langle 1, =_x, =_x^r\rangle\langle 4, >_x, \bowtie_x^r\rangle\langle 5, =_x, =_x^r\rangle\}^\omega$. The refined abstraction is empty, hence the program has no infinite runs.

Remark. In light of the fact that the universal halting problem in not r.e., in general, the abstraction-refinement loop is not guaranteed to terminate, even if the program terminates.

4 PROVING TERMINATION OF PROGRAMS WITH TREES

In this section we instantiate our termination verification framework for programs manipulating tree-like data structures. We consider sequential, non-recursive C-like programs working over tree-shaped data structures with a finite set of pointer variables $PVar$. Each node in a tree contains a `data` value field, ranging over a finite set $Data$ and three selector fields, denoted `left`, `right`, and `up`.[7] For $x, y \in PVar$ and $d \in Data$, we consider the programs over the set of instructions $\mathfrak{I}_T$ composed of the following:

- **guards**: $x == \texttt{null}$, $x == y$, $x.\texttt{data} == d$, and Boolean combinations of the above,
- **actions**: $x = \texttt{null}$, $x = y$, $x = y.\{\texttt{left}|\texttt{right}|\texttt{up}\}$, $x.\texttt{data} = d$, $x.\{\texttt{left}|\texttt{right}\} = \texttt{new}$ and $x.\{\texttt{left}|\texttt{right}\} = \texttt{null}$.

```
0   x := root;
1   while (x!=null)
2       if (x.left!=null) and
            (x.left.data!=mark)
3           x:=x.left;
4       else if (x.right!=null) and
            (x.right.data!=mark)
5           x:=x.right;
        else
6           x.data:=marked;
7           x:=x.up;
```

Figure 9. Depth-first tree traversal

This set of instructions covers a large class of practical tree-manipulating procedures[8]. For instance, Figure 9 shows a depth-first tree traversal procedure, com-

[7] Generalizing to trees with another arity is straightforward.

[8] Handling general updates of the form $x.\texttt{left}|\texttt{right} := y$ is more problematic, since in general the result of such instructions is not a tree any more. Up to some extent, this

monly used in real-life programs. In particular, here $PVar = \{x\}$ and $Data = \{marked, unmarked\}$. This program will be used as a running example in the rest of the section.

In order to use our framework for analyzing termination of programs with trees, we need to provide (1) well-founded partial orderings on the tree domain, (2) symbolic encodings for the partial orderings as well as for the program semantics and (3) a decision procedure for the spuriousness problem. The last point was tackled in our previous work [20], for lassos without destructive updates (i.e., instructions of the form $x.\texttt{left}|\texttt{right} := \texttt{new}|\texttt{null}$).

4.1 Abstracting Programs with Trees into Büchi Automata

A memory configuration is a binary tree with nodes labeled by elements of the set $\mathcal{C} = Data \times 2^{PVar} \cup \{\square\}$, i.e., a node is either null ($\square$) or it contains a data value and a set of pointer variables pointing to it ($\langle d, V\rangle \in D \times 2^{PVar}$). Each pointer variable can point to at most one tree node (if it is null, it does not appear in the tree). For a tree $t \in \mathcal{T}(\mathcal{C})$ and a position $p \in dom(t)$ such that $t(p) = \langle d, V\rangle$, we denote $\delta_t(p) = d$ and $\nu_t(p) = V$. For clarity reasons, the semantics of the program instructions considered is given in Figure 10. First we show that all program actions considered here can be encoded as rational tree relations.

$$\frac{\forall p \in dom(t) : x \notin \nu(t(p))}{\langle l, t\rangle \to \langle l', t\rangle} \quad \frac{p \in dom(t) \quad x \in \nu(t(p))}{\langle l, t\rangle \to \langle l', t[p \leftarrow \langle \delta(t(p)), \nu(t(p)) \setminus \{x\}\rangle]\rangle} \; x = \texttt{null}$$

$$\frac{p \in dom(t) \quad y \in \nu(t(p)) \quad t(p.0) \neq \square \quad \langle l, t\rangle \xrightarrow{x=null} \langle l', t'\rangle}{\langle l, t\rangle \to \langle l', t'[p.0 \leftarrow \langle \delta(t(p.0)), \nu(t(p.0)) \cup \{x\}\rangle]\rangle} \; x = y.\texttt{left}(a)$$

$$\frac{p \in dom(t) \quad y \in \nu(t(p)) \quad t(p.0) = \square \quad \langle l, t\rangle \xrightarrow{x=null} \langle l', t'\rangle}{\langle l, t\rangle \to \langle l', t'\rangle} \quad \frac{\forall p \in dom(t).y \notin \nu(t(p))}{\langle l, t\rangle \to Err} \; x = y.\texttt{left}(b)$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p))}{\langle l, t\rangle \to \langle l', t[p \leftarrow \langle d, \nu(t(p))\rangle]\rangle} \quad \frac{\forall p \in dom(t).x \notin \nu(t(p))}{\langle l, t\rangle \to Err} \; x.\texttt{data} = d$$

$$\frac{d \in Data \quad x \in \nu(t(p)) \quad p \in dom(t) \quad t(p.0) = \square}{\langle l, t\rangle \to \langle l', t[p.0 \leftarrow \langle d, \emptyset\rangle, p.0.0 \leftarrow \square, p.0.1 \leftarrow \square]\rangle} \quad \frac{(\forall p \in dom(t).x \notin \nu(t(p))) \vee (p \in dom(t) \wedge x \in \nu(t(p)) \wedge t(p.0) \neq \square)}{\langle l, t\rangle \to Err} \; x.\texttt{left} = new$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p)) \quad p.0 \in dom(t) \quad t(p.0.0) = t(p.0.1) = \square}{\langle l, t\rangle \to \langle l', t[p.0 \leftarrow \square, p.0.0 \leftarrow \bot, p.0.1 \leftarrow \bot]\rangle} \quad \frac{\begin{array}{l}(\forall p \in dom(t).x \notin \nu(t(p))) \vee \\ (p \in dom(t) \wedge x \in \nu(t(p)) \wedge \\ \quad (t(p.0) = \square \vee t(p.0.0) \neq \square \vee t(p.1) \neq \square))\end{array}}{\langle l, t\rangle \to Err} \; x.\texttt{left} = null$$

Figure 10. The concrete semantics of program statements – the upper part of a rule represents a guard and the lower part an action. *Err* is equal to abnormal termination of the program (it is not possible to execute the given statement).

kind of programs can be handled by the extension described in Section 4.3.

Lemma 6. For any program instruction $i = \langle g, a \rangle \in \mathfrak{I}_T$, the tree relation $R_i = \{\langle t, t' \rangle | t \in g, t' = a(t)\}$ is rational.

Proof. We will now show, how to construct a tree automaton for each guard and each statement. The automaton for the guarded action is then created as composition of the corresponding guard and the action. Both the guards and actions are tree automata $A = (Q, F, \Delta)$ over the pair alphabet $(\mathcal{C} \cup \{\Box, \bot\}) \times (\mathcal{C} \cup \{\Box, \bot\})$. We use the following subsets of the alphabet $\mathcal{C}$ within the proof:

- $N_V = \{\langle d, X \rangle \in C | \forall x \in V . x \notin X\}$
- $P_V = \{\langle d, X \rangle \in C | \forall x \in V . x \in X\}$

guards

- $x == \texttt{null}$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\forall p \in N_{\{x\}} \langle p, p \rangle (q_1, q_1) \to q_1$

  $\}$
- $x == y$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\forall p \in N_{\{x,y\}} \langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall p \in P_{\{x,y\}} \langle p, p \rangle (q_1, q_1) \to q_1$

  $\}$
- $x.\texttt{data} == d$: $A = (\{q_1, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\forall p \in N_{\{x\}} . \langle p, p \rangle (q_1, q_1) \to q_1\}$
  - $\forall \langle d, X \rangle \in P_{\{x\}} . \langle \langle d, X \rangle, \langle d, X \rangle \rangle (q_1, q_1) \to q_2$
  - $\forall p \in N_{\{x\}} \langle p, p \rangle (q_2, q_1) \to q_2$ and $\langle p, p \rangle (q_1, q_2) \to q_2$

  $\}$

actions

- $x = \texttt{null}$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\forall p \in N_{\{x\}} \langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d, X \rangle \in P_{\{x\}} \langle \langle d, X \rangle, \langle d, X \setminus \{x\} \rangle \rangle (q_1, q_1) \to q_1$

  $\}$

- $y = x$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \square, \square \rangle \to q_1$
  - $\forall p \in N_{\{x,y\}}.\langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d, X \rangle \in P_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \cup \{y\} \rangle \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d, X \rangle \in \{P_{\{y\}} \setminus P_{\{x\}}\}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_1, q_1) \to q_1$

  $\}$
- $y = x.\texttt{left}$, $A = (\{q_1, q_y, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \square, \square \rangle \to q_1$
  - $\langle \square, \square \rangle \to q_y$
  - $\forall \langle d, X \rangle \in N_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \cup \{y\} \rangle \rangle (q_1, q_1) \to q_y$
  - $\forall \langle d, X \rangle \in N_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d, X \rangle \in P_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_y, q_1) \to q_2$
  - $\forall \langle d, X \rangle \in N_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_2, q_1) \to q_2$
  - $\forall \langle d, X \rangle \in N_{\{x\}}.\langle \langle d, X \rangle, \langle d, X \setminus \{y\} \rangle \rangle (q_1, q_2) \to q_2$

  $\}$
- $x.\texttt{data} = d$: $A = (\{q_1, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \square, \square \rangle \to q_1$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall \langle d_{orig}, X \rangle \in P_{\{x\}}.\langle \langle d_{orig}, X \rangle, \langle d, X \rangle \rangle (q_1, q_1) \to q_2$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_2, q_1) \to q_2$
  - $\langle p, p \rangle (q_1, q_2) \to q_2$

  $\}$
- $x.\texttt{left} = \texttt{new}$: $A = (\{q_1, q_{new}, q_\perp, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \square, \square \rangle \to q_1$
  - $\langle \perp, \square \rangle \to q_\perp$
  - $\langle \square, \langle d_{init}, \emptyset \rangle \rangle (q_\perp, q_\perp) \to q_{new}$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_1, q_1) \to q_1$
  - $\forall p \in P_{\{x\}} \langle p, p \rangle (q_{new}, q_1) \to q_2$
  - $\forall p \in N_{\{x\}}.\langle p, p \rangle (q_2, q_1) \to q_2$
  - $\langle p, p \rangle (q_1, q_2) \to q_2$

  $\}$
- $x.\texttt{left} = \texttt{null}$. $A = (\{q_1, q_{del}, q_\perp, q_2\}, \{q_2\}, \Delta)$ with $\Delta = \{$
  - $\langle \square, \square \rangle \to q_1$
  - $\langle \square, \perp \rangle \to q_\perp$
  - $\forall p \in N_{\{x\}} \langle p, \square \rangle (q_\perp, q_\perp) \to q_{del}$

- $\forall p \in N_{\{x\}}.\langle p, p\rangle(q_1, q_1) \to q_1$
- $\forall p \in P_{\{x\}}\langle p, p\rangle(q_{del}, q_1) \to q_2$
- $\forall p \in N_{\{x\}}.\langle p, p\rangle(q_2, q_1) \to q_2$
- $\langle p, p\rangle(q_1, q_2) \to q_2$

}

Instructions $y = x.\texttt{right}$ and $y = x.\texttt{up}$ are similar to the $y = x.\texttt{left}$, the instruction $x.\texttt{right} = \texttt{new}$ to the $x.\texttt{left} = \texttt{new}$, and the instruction $x.\texttt{right} = \texttt{null}$ to the $x.\texttt{left} = \texttt{null}$, □

In order to abstract programs with trees as Büchi automata (cf. Definition 12), we must introduce the well-founded partial orders on the working domain, i.e., trees. These well founded orders are captured by Definition 14, and the working domain will be $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq^r_x\}_{x \in PVar}, \{\preceq_d, \preceq^r_d\}_{d \in Data}\rangle$.

Definition 14 (Well-founded orders on trees)**.**

- $t_1 \preceq_x t_2$, for some $x \in PVar$ iff

  1. $dom(t_1) \subseteq dom(t_2)$, and
  2. there exist positions $p_1 \in dom(t_1)$, $p_2 \in dom(t_2)$ such that $x \in \nu_{t_1}(p_1)$, $x \in \nu_{t_2}(p_2)$ and $p_1 \leq_{lex} p_2$.

  In other words t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 and the position of x in t_1 is lexicographically smaller than the position of x in t_2.

- $t_1 \preceq^r_x t_2$, for some $x \in PVar$ iff

  1. $dom(t_1) \subseteq dom(t_2)$, and
  2. there exist positions $p_1 \in dom(t_1)$, $p_2 \in dom(t_2)$ such that $x \in \nu_{t_1}(p_1)$, $x \in \nu_{t_2}(p_2)$ and $p_1 \geq_{lex} p_2$.

  In other words, t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 and the position of x in t_1 is lexicographically bigger than the position of x in t_2.

- $t_1 \preceq_d t_2$, for some $d \in Data$ iff for any position $p \in dom(t_1)$ such that $\delta_{t_1}(p) = d$ we have $p \in dom(t_2)$ and $\delta_{t_2}(p) = d$. In other words, t_1 is smaller than t_2 if the set of nodes colored with d in t_1 is a subset of the set of nodes colored with d in t_2.

- $t_1 \preceq^r_d t_2$, for some $d \in Data$ iff

  1. $dom(t_1) \subseteq dom(t_2)$, and
  2. for any position $p \in dom(t_2)$ such that $\delta_{t_2}(p) = d$ we have $p \in dom(t_1)$ and $\delta_{t_1}(p) = d$.

  In other words, t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 and the set of nodes colored with d in t_2 is a subset of the set of nodes colored with d in t_1.

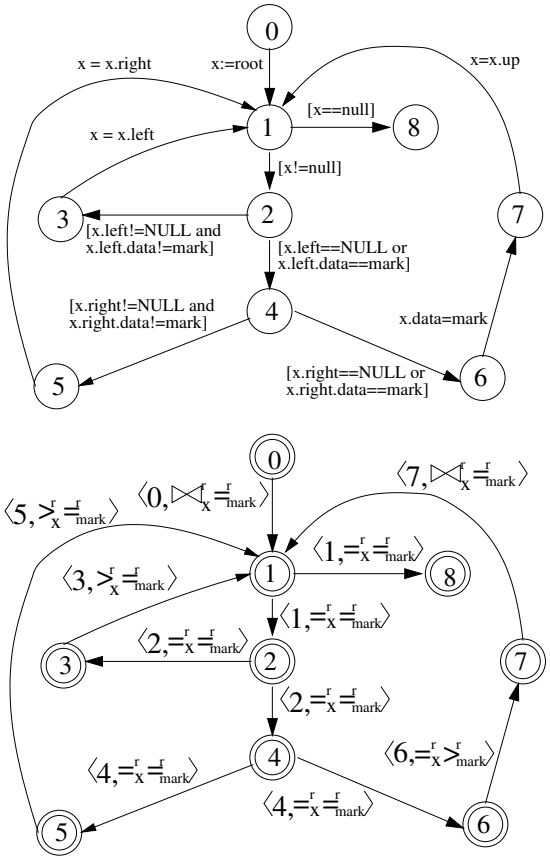


Figure 11. The depth-first tree traversal procedure and its initial abstraction

Lemma 7. The relations $\preceq_x$, $\preceq_x^r$, $x \in PVar$ and $\preceq_d$,$\preceq_d^r$, $d \in Data$ are well-founded.

Proof.

- Let us suppose that there exists an infinite sequence of trees $t_0 \succ_x t_1 \succ_x t_2 \succ_x \ldots$ Then according to the definition of $\succeq_x$: $\forall i \geq 0\ \exists p_i \in dom(t_i)$ such that $x \in \nu_{t_i}(p_i)$ and $p_i \geq_{lex} p_{i+1}$ and $dom(t_i) \supseteq dom(t_{i+1})$. Moreover, due to the fact that $t_{i+1} \not\preceq_x t_i$, one of the following holds for each $i \geq 0$: (i) $p_i >_{lex} p_{i+1}$ or (ii) $dom(t_i) \supset dom(t_{i+1})$. Note that at least one of the cases (i) or (ii) holds for infinitely many different values of i.

  Therefore in the infinite sequence of trees $t_0 \succ_x t_1 \succ_x t_2 \succ_x \ldots$, there must exist an infinite subsequence[9] $\overline{t_0} \succ_x \overline{t_1} \succ_x \overline{t_2} \succ_x \ldots$ such that either (i) or (ii) holds for all $i \geq 0$ in this sequence.

  Let us suppose that we have the infinite sequence $\overline{t_0} \succ_x \overline{t_1} \succ_x \overline{t_2} \succ_x \ldots$ where

[9] An infinite sequence $a_0, a_1, a_2, \ldots$ is a subsequence of the infinite sequence $b_0, b_1, b_2, \ldots$ iff exists a mapping $\sigma : \mathbb{N} \to \mathbb{N}$ such that $\forall i \geq 0$. $a_i = b_{\sigma(i)}$ and $i < j \Rightarrow \sigma(i) < \sigma(j)$.

the condition (i) holds for all $i \geq 0$. Note that for all $i \geq 0.dom(\overline{t_i}) \subseteq dom(\overline{t_0})$ and $dom(\overline{t_0})$ is finite. Therefore there exists only a *finite number of different positions* $\overline{p_0}, \overline{p_1}, \ldots, \overline{p_k}$, hence the sequence $\overline{t_0} \succ_x \overline{t_1} \succ_x \overline{t_2} \succ_x \ldots$ cannot be infinite.

Let us suppose that the condition (ii) holds for all $i \geq 0$. Then there must be an infinite sequence $dom(\overline{t_0}) \supset dom(\overline{t_1}) \supset dom(\overline{t_2}) \supset \ldots$. The $dom(\overline{t_0})$ is finite, hence the sequence cannot exist.

The proof of well-foundedness of $\preceq^r_x$ is similar.

- Let us suppose that for some $d \in Data$ there exists an infinite sequence $t_0 \succ_d t_1 \succ_d t_2 \succ_d \ldots$ Let $S_d(t) = \{p \in dom(t) \mid \delta_t(p) = d\}$ be a set of all positions labeled by a data value d in the tree t. Then according to the definition of $\preceq_d$: $S_d(t_0) \supset S_d(t_1) \supset S_d(t_2) \supset \ldots$ The $dom(t_0)$ is finite, hence the set $S_d(t_0)$ is also finite. Therefore the sequence $t_0 \succ_d t_1 \succ_d t_2 \succ_d \ldots$ must be finite.
- Let us suppose that for some $d \in Data$ there exists an infinite sequence $t_0 \succ^r_d t_1 \succ^r_d t_2 \succ^r_d \ldots$ Let $S_d(t) = \{p \in dom(t) \mid \delta_t(p) = d\}$ be a set of all positions labeled by a data value d in the tree t. Then according to the definition of $\preceq^r_d$: $S_d(t_0) \subseteq S_d(t_1) \subseteq S_d(t_2) \subseteq \ldots$ and $dom(t_0) \supseteq dom(t_1) \supseteq dom(t_2) \supseteq \ldots$ Moreover, due to the fact that $\forall i \geq 0.\ t_{i+1} \not\preceq_x t_i$, one of the following situations holds for each value of i:

  1. $S_d(t_i) \subset S_d(t_{i+1})$ or
  2. $dom(t_i) \supset dom(t_{i+1})$.

  Therefore in the infinite sequence of trees $t_0 \succ^r_d t_1 \succ^r_d t_2 \succ^r_d \ldots$, there must exist an infinite subsequence $\overline{t_0} \succ^r_d \overline{t_1} \succ^r_d \overline{t_2} \succ^r_d \ldots$ such that either (i) or (ii) holds for all $i \geq 0$ in this sequence.

  1. cannot be true in the whole infinite sequence, because $\forall i > 0.\ S_i \subseteq dom(t_0)$ and $dom(t_0)$ is finite.
  2. cannot be true, because $dom(t_0)$ is finite.

  Hence the sequence $t_0 \succ^r_d t_1 \succ^r_d t_2 \succ^r_d \ldots$ must be finite.

□

Lemma 7 implies that Assumption 1 is valid for the working domain $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq^r_x\}_{x \in PVar}, \{\preceq_d, \preceq^r_d\}_{d \in Data} \rangle$, and hence the whole termination analysis framework presented in Section 3 can be employed.

The choice of relations is ad-hoc for the time being. In practice these relations are sufficient for proving termination of an important class of programs handling trees.

Lemma 8. The relations $\preceq_x$, $\preceq^r_x$, $x \in PVar$ and $\preceq_d$,$\preceq^r_d$, $d \in Data$ are rational.

Proof. By the construction of tree automata $A = (Q, F, \Delta)$ over the pair alphabet $(\mathcal{C} \cup \{\Box, \bot\}) \times (\mathcal{C} \cup \{\Box, \bot\})$ for each of the proposed relations.

- $t_1 \preceq_x t_2$: $A = (\{q_1, q_L, q_R, q_{acc}\}, \{q_{acc}\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\langle \bot, \Box \rangle \to q_1$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_1, q_1) \to q_1$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_L, q_1) \to q_L$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_1, q_L) \to q_L$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_R, q_1) \to q_R$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_1, q_R) \to q_R$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_x, p \rangle (q_1, q_1) \to q_L$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p, p_x \rangle (q_1, q_1) \to q_R$
  - $\forall p_1, p_2 \in P_{\{x\}}.\langle p_1, p_2 \rangle (q_1, q_1) \to q_{acc}$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}.\langle p_x, p \rangle (q_R, q_1) \to q_{acc}$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}.\langle p_x, p \rangle (q_1, q_R) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2 \rangle (q_L, q_R) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2 \rangle (q_{acc}, q_1) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2 \rangle (q_1, q_{acc}) \to q_{acc}$

  $\}$
- $t_1 \preceq_x^r t_2$: $A = (\{q_1, q_L, q_R, q_{acc}\}, \{q_{acc}\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\langle \bot, \Box \rangle \to q_1$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_1, q_1) \to q_1$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_L, q_1) \to q_L$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_1, q_L) \to q_L$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_R, q_1) \to q_R$
  - $\forall p_1, p_2 \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_1, p_2 \rangle (q_1, q_R) \to q_R$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p_x, p \rangle (q_1, q_1) \to q_L$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}} \cup \{\Box, \bot\}.\langle p, p_x \rangle (q_1, q_1) \to q_R$
  - $\forall p_1, p_2 \in P_{\{x\}}.\langle p_1, p_2 \rangle (q_1, q_1) \to q_{acc}$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}.\langle p, p_x \rangle (q_L, q_1) \to q_{acc}$
  - $\forall p_x \in P_{\{x\}} \forall p \in N_{\{x\}}.\langle p, p_x \rangle (q_1, q_L) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2 \rangle (q_R, q_L) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2 \rangle (q_{acc}, q_1) \to q_{acc}$
  - $\forall p_1, p_2 \in N_{\{x\}}.\langle p_1, p_2 \rangle (q_1, q_{acc}) \to q_{acc}$

  $\}$
- $t_1 \preceq_d t_2$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box \rangle \to q_1$
  - $\langle \bot, \Box \rangle \to q_1$
  - $\langle \Box, \bot \rangle \to q_1$

  - $\forall p_1, p_2 \in \mathcal{C} \cup \{\bot\}, p_1 \neq \langle d, X\rangle$ for some $X \in 2^{PVar}$ $\langle p_1, p_2\rangle(q_1, q_1) \rightarrow q_1$.
  - $\forall X_1, X_2 \in 2^{PVar}.\langle\langle d, X_1\rangle, \langle d, X_2\rangle\rangle(q_1, q_1) \rightarrow q_1$

  }
- $t_1 \preceq^r_d t_2$: $A = (\{q_1\}, \{q_1\}, \Delta)$ with $\Delta = \{$
  - $\langle \Box, \Box\rangle \rightarrow q_1$
  - $\langle \bot, \Box\rangle \rightarrow q_1$
  - $\forall p_1, p_2 \in \mathcal{C} \cup \{\bot\}, p_2 \neq \langle d, X\rangle$ for some $X \in 2^{PVar}$ $\langle p_1, p_2\rangle(q_1, q_1) \rightarrow q_1$.
  - $\forall X_1, X_2 \in 2^{PVar}.\langle\langle d, X_1\rangle, \langle d, X_2\rangle\rangle(q_1, q_1) \rightarrow q_1$

  }

□

The Büchi automaton representing the initial abstraction of the depth-first tree traversal procedure is depicted in Figure 11. To simplify the figure, we use only the orders $\preceq^r_x$ and $\preceq^r_{mark}$. Thanks to these orders, there is no potential infinite run in the abstraction.

Remark. In practice, one often obtains a more precise initial abstraction by composing several program steps into one. In cases where the program instructions induce rational tree relations, it is guaranteed that composition of two or more program steps also induces a rational tree relation.

In order to decide which transitions will be composed, one can use the concept of so-called cutpoints. Formally, given a program $P = \langle \mathfrak{I}, L, l_0, \Rightarrow\rangle$, a set of cutpoints $S \subseteq L$ is a set of control locations such that each loop in the program contains at least one location $l \in S$ [12, 1]. The set of cutpoints can be provided manually or discovered automatically by means of some heuristics. In our heuristics we provide one cutpoint on each branch in the control flow.

4.2 Adding Tree Rotations

As a possible extension of the proposed framework, one can allow tree left and right rotations as program statements [15]. The effect of a left tree rotation on a node pointed by variable x is depicted in Figure 12 (the effect of the right rotation is analogous). The concrete semantics of the rotations is depicted in Figure 13, where $u_{left} = t_\epsilon\{\lambda \leftarrow t_{|p.1}\}\{0 \leftarrow t_{|p}\}\{0.1 \leftarrow t_{|p.1.0}\}$, and $u_{right} = t_\epsilon\{\lambda \leftarrow t_{|p.0}\}\{1 \leftarrow t_{|p}\}\{1.0 \leftarrow t_{|p.0.1}\}$.

Since rotations cannot be described by rational tree relations, we cannot check whether $\preceq_x, \preceq^r_x, \preceq_d$ and $\preceq^r_d$ hold, simply by intersection. However, we know that rotations do not change the number of nodes in the tree, therefore we can label

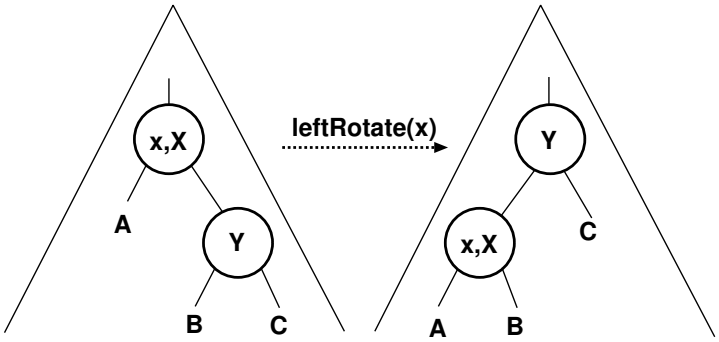


Figure 12. Left tree rotation on the node pointed by variable x. X, and Y denote two concrete nodes affected by the rotation and A, B, and C three subtrees.

$$\frac{p \in dom(t) \quad x \in \nu(t(p)) \quad t(p.1) \neq \square}{\langle l, t\rangle \rightarrow \langle l', t\{p \leftarrow u_{left}\}\rangle} \qquad \frac{\begin{array}{l}(\forall p \in dom(t).x \notin \nu(t(p))) \vee \\ (p \in dom(t) \wedge x \in \nu(t(p)) \wedge t(p.1) = \square)\end{array}}{\langle l, t\rangle \rightarrow Err}\ \texttt{leftRotate}(x)$$

$$\frac{p \in dom(t) \quad x \in \nu(t(p)) \quad t(p.0) \neq \square}{\langle l, t\rangle \rightarrow \langle l', t\{p \leftarrow u_{right}\}\rangle} \qquad \frac{\begin{array}{l}(\forall p \in dom(t).x \notin \nu(t(p))) \vee \\ (p \in dom(t) \wedge x \in \nu(t(p)) \wedge t(p.0) = \square)\end{array}}{\langle l, t\rangle \rightarrow Err}\ \texttt{rightRotate}(x)$$

Figure 13. Concrete semantics of tree rotations. As in the case of Figure 10, the upper part of a rule represents a guard and the lower part of a rule represents an action on a tree. *Err* is equal to abnormal termination of the program (the rotation is not possible).

them a posteriori with $=_d$,$=^r_d$, $d \in Data$, and $\bowtie_x$, $\bowtie^r_x$, $x \in PVar$, since the relative positions of the variables after the rotations are not known[10]. This extension has been used to verify termination of the *Red-black delete* and *Red-black insert* examples reported in Section 5.

4.2.1 Example: Red-black Trees – Rebalancing After Delete

Red-black trees [15] are binary search trees with the following *red-black* balanceness properties:

1. Every node carries an extra flag set either to *red*, or *black*.
2. The root of the tree is black.
3. Every leaf node (node without a child) is black.
4. If a node is red, then both its children are black.

[10] We could make this abstraction more precise by labeling with $=_x$, $=^r_x$ for all $x \in PVar$ situated *above* the rotation point – the latter condition can be checked by intersection with a rational tree language.

5. For each node in the tree, all simple paths from this node to the leaves have equal number of black nodes.

After deletion of a node from such a tree, the balance property can be broken. In order to restore the red-black properties, the rebalance procedure displayed in Figure 15 is executed. In order to simplify the presentation of our termination analysis framework, we choose a set of cutpoints from the original program. The initial abstraction using these cutpoints is depicted in Figure 14. We show only the relation $\preceq_x$, which is important for the termination proof. Note that there is no edge $14 \to 3$ and $35 \to 3$ in the abstraction, because there is no execution of the program following these edges.

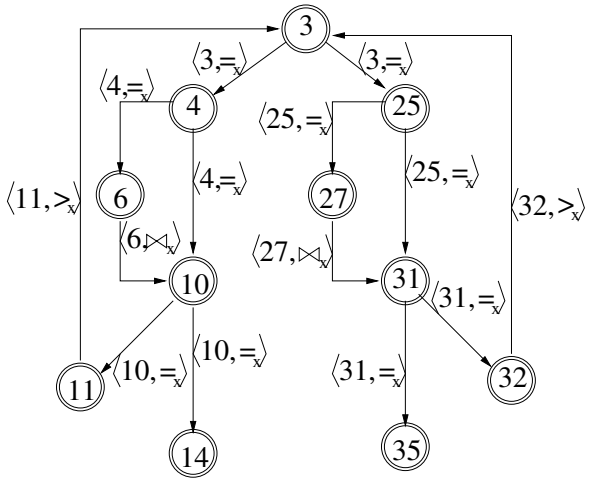


Figure 14. Red-Black: rebalance after delete - initial abstraction on the simplified control flow

It can be seen in Figure 14 that there are counterexamples of the following forms:

1. $\Sigma^*(\Sigma^*\langle 3, =_x\rangle\langle 4, =_x\rangle\langle 6, \bowtie_x\rangle\langle 10, =_x\rangle\langle 11, >_x\rangle\Sigma^*)^\omega$
2. $\Sigma^*(\Sigma^*\langle 3, =_x\rangle\langle 25, =_x\rangle\langle 27, \bowtie_x\rangle\langle 31, =_x\rangle\langle 32, >_x\rangle\Sigma^*)^\omega$

The abstraction refinement works as follows. First, the counterexample $(\langle 3, =_x\rangle \langle 4, =_x\rangle\langle 6, \bowtie_x\rangle\langle 10, =_x\rangle\langle 11, >_x\rangle)^\omega$ is taken. There is no execution of the program following the path 3, 4, 6, 10, 11, 3 in the program, so we can apply Lemma 4 and remove the whole set of counterexamples $\Sigma^*\langle 3, \_\rangle\langle 4, \_\rangle\langle 6, \_\rangle\langle 10, \_\rangle\langle 11, \_\rangle(\Sigma^*)^\omega$. Note that all counterexamples of type 1 are included. The next counterexample is $(\langle 3, =_x\rangle\langle 25, =_x\rangle\langle 27, \bowtie_x\rangle\langle 31, =_x\rangle\langle 32, >_x\rangle)^\omega$. As for the previous one, there is no execution in the program following the path 3, 25, 27, 31, 32, 3 so according to Lemma 4, we can remove the counterexamples $\Sigma^*\langle 3, \_\rangle\langle 25, \_\rangle\langle 27, \_\rangle\langle 31, \_\rangle\langle 32, \_\rangle(\Sigma^*)^\omega$ (all counterexamples of type 2 are included).

After this second refinement, there is no counterexample left in the abstraction, so we can conclude that the Red-Black rebalance procedure terminates.

```
    procedure rb_rebalance(root)
0   while true
1       if x==root then return;
2       if x.color=RED return;
3       if x==x.parrent.left then
4           w=x.parrent.right
5           if w.color==red
6               w.color=black
7               x.parrent.color=red
8               leftrotate(x.parent)
9               w=x.parrent.right
10          if (w.left==null or w.left.color==black) and
                (w.right==null or w.right.color==black) then
11              w.color=red
12              x=x.parent
13              continue
14          if (w.right==null or w.right.color==black) then
15              w.left.color=black
16              w.color=red
17              rightrotate(w)
18              w=x.parrent.right
19          w.color=px.color
20          x.parrent.color=black
21          w.right.color=black
22          leftrotate(x.parent)
23          x=root
24      else
25          w=x.parrent.left
26          if w.color==red
27              w.color=black
28              x.parrent.color=red
29              rightrotate(x.parent)
30              w=x.parrent.left
31          if (w.left==null or w.left.color==black) and
                (w.right==null or w.right.color==black) then
32              w.color=red
33              x=x.parent
34              continue
35          if (w.left==null or w.left.color==black) then
36              w.right.color=black
37              w.color=red
38              leftrotate(w)
39              w=x.parrent.left
40          w.color=x.parrent.color
41          x.parrent.color=black
42          w.left.color=black
43          rightrotate(x.parent)
44          x=root
```

Figure 15. Red-Black: rebalance after delete

4.3 From Trees to Complex Data Structure

In general, programs with pointer data types do not manipulate just lists and trees. The structures that occur at execution time can be arbitrarily oriented graphs. In this section we use the termination detection framework for programs that manipulate such complex structures. To compute program invariants, we use the approach from [6], based on encoding of graphs as trees with extra edges. The basic idea of this encoding is that each structure has an underlying tree (called a *backbone*) which remains unchanged during the whole computation.

As in the previous, let $PVar$ be a finite set of pointer variables, $Data$ a finite set of data values, N the maximal number of selectors allowed in a single memory node, and S a finite set of *pointer descriptors*, which are references to regular expressions (called routing expressions) over the alphabet of directions in the tree (e.g., left, right, left-up, right-up, etc.). The backbone is a tree labeled by symbols of the alphabet $\mathcal{C} = (Data \times 2^{PVar} \times S^N) \cup \{\Box, \Diamond\}$. The arity function is defined as follows: $\#(\Box) = 0$ and $\#(c) = b$ for all $c \in \mathcal{C} \setminus \{\Box\}$, where $b > 0$ is a branching factor fixed for the concrete encoding[11]. Each node of such a tree is either *active*, *removed*, or *unused*, as follows:

active nodes represent memory cells present in the memory. These nodes are labeled by symbols from the alphabet $Data \times 2^{PVar} \times S^N$;

removed nodes are the nodes which were deleted from the memory. They are labeled by the symbol $\Diamond$. These nodes are kept in the data structure in order to preserve original shape of the underlying tree;

unused nodes are labeled by $\Box$ and they are placed in leaves. These nodes can be changed into active ones by a `new` statement.

Pointers between the active memory nodes are represented by pointer descriptors (placed in the active nodes). A pointer descriptor corresponding to $1 \leq i \leq N$ determines the destination of the i^{th} selector field of the node. There are two special designated pointer descriptors for the *null* and *undefined* pointers. The destructive updates on the data structure can be then performed simply by changing the pointer descriptors inside the tree followed by an update of the corresponding routing expression.

A set of such data structures (*trees with routing expressions*) can be represented by tree automata. This allows to apply the framework of *abstract regular tree model checking* (ARTMC) [5] in order to compute (over-approximations of) the invariants of programs handling complex data structures. The ARTMC tool derives automatically the set of underlying trees and the corresponding routing expressions.

We apply the termination analysis framework along the same lines as in the previous. For a tree $t \in \mathcal{T}(\mathcal{C})$ and a position $p \in dom(t)$ such that $t(p) = \langle d, V, s_1, \ldots s_N \rangle$, we denote $\delta_t(p) = d$, $\nu_t(p) = V$, and $\xi_t(p)[i] = s_i$. If $t(p) = \Diamond$,

[11] Usually, the branching factor b is equal to the number of selectors N.

$\delta_t(p) = \bot$, $\nu_t(p) = \emptyset$, and $\xi_t(p)[i] = \mathit{undefined}, 1 \leq i \leq N$, i.e., all descriptors represent *undefined* pointers.

Now, we need to provide well-founded orders on trees with routing expressions. We use the fact that the tree backbone is not changed during the whole computation, hence we can easily employ the orders $\preceq_x$, $\preceq^r_x$, $\preceq_d$ and $\preceq^r_d$ defined in Definition 14. In addition, we define the following well-founded orders based on pointer descriptors values captured by Definition 15. The working domain will be $D_T = \langle \mathcal{T}(\mathcal{C}), \{\preceq_x, \preceq^r_x\}_{x \in PVar}, \{\preceq_d, \preceq^r_d\}_{d \in Data}, \{\preceq_{1:s}, \preceq^r_{1:s}\}_{s \in S}, \ldots, \{\preceq_{N:s}, \preceq^r_{N:s}\}_{s \in S} \rangle$.

Definition 15 (Well-founded orders based on pointer descriptors)**.**

- $t_1 \preceq_{i:s} t_2$, for some $1 \leq i \leq N$ and $s \in S$ iff for any position $p \in \mathit{dom}(t_1)$ such that $\xi_{t_1}(p)[i] = s$ we have $p \in \mathit{dom}(t_2)$ and $\xi_{t_2}(p)[i] = s$. In other words, t_1 is smaller than t_2 if the set of nodes in t_1, where the i^{th} descriptor is set to s, is a subset of the set of nodes in t_2, where the i^{th} descriptor is set to s.
- $t_1 \preceq^r_{i:s} t_2$, for some $1 \leq i \leq N$ and $s \in S$ iff (i) $\mathit{dom}(t_1) \subseteq \mathit{dom}(t_2)$ and (ii) for any position $p \in \mathit{dom}(t_2)$ such that $\xi_{t_2}(p)[i] = s$ we have $p \in \mathit{dom}(t_1)$ and $\xi_{t_1}(p)[i] = s$. In other words t_1 is smaller than t_2 if all nodes in t_1 are also present in t_2 (i.e., no new nodes were created) and the set of nodes in t_2, where the i^{th} descriptor is set to s, is a subset of the set of nodes in t_1, where the i^{th} descriptor is set to s.

Lemma 9. The relations on pointer descriptor fields $\preceq_{i:s}$ and $\preceq^r_{i:s}$ are well-founded and rational.

Proof. The pointer descriptor is syntactically a data value from a finite set S. Therefore the proofs of rationality and well-foundedness of $\preceq_{i:s}$ (resp $\preceq^r_{i:s}$) are similar to the proofs of $\preceq_d$ (resp. $\preceq^r_d$). □

To understand the use of the relations $\preceq_{i:s}, \preceq^r_{i:s}$, consider the program from Figure 16. This procedure traverses a binary tree in depth-first order and links all its nodes into a cyclic singly-linked list using the selector *next*. In the beginning, all *next* selector are set to null. Then the processed nodes have the selector *next* non-null. Due to this fact, one can establish termination proof using the orders $\preceq^R_x$ and $\preceq_{next:null}$. Note that the order based on data values (as in classical depth-first traversal) cannot be used any more to establish termination.

5 IMPLEMENTATION AND EXPERIMENTAL RESULTS

We have implemented a prototype tool that uses this framework to detect termination of programs with trees and trees with extra edges. The tool was built as an extension of the ARTMC [6] verifier for safety properties (null-pointer dereferences, memory leaks, etc.). We applied our tool to several programs that manipulate.

doubly-linked lists: *DLL-insert* (*DLL-delete*) which inserts (deletes) a node in (from) a doubly-linked list, and *DLL-reverse* which is the in-place reversal of the doubly linked list;

```
    procedure link_nodes(root)
0   x := root;
1   last:=root;
2   while (x!=null)
3       if (x.left!=null) and (x.left.next==null)
4           x:=x.left;
5       else if (x.right!=null) and (x.right.next==null)
6           x:=x.right;
        else
7           x.next:=last;
8           last:=x;
9           x:=x.up;
```

Figure 16. Linking nodes in Depth-first order

Example	Time	N_{refs}
DLL-insert	2s	0
DLL-delete	1s	0
DLL-reverse	2s	0
Depth-first search	17s	0
Linking leaves in trees	14s	0
Deutsch-Schorr-Waite	1m 24s	0
Linking Nodes	5m 47s	0
Red-black delete	4m 54s	2
Red-black insert	29s	0

Table 1. Experimental results

trees: *Depth-first search* and *Deutsch-Schorr-Waite* which are tree traversals, *Red-black delete* (*insert*) which rebalances a red-black tree after the deletion (insertion) of a node;

tree with extra edges: *Linking leaves* (*Linking nodes*) which insert all leaves (nodes) of a tree in a singly-linked list.

The results obtained on an Intel Core 2 PC with 2.4 GHz CPU and 2 GB RAM memory are given in Table 1. The field *time* represents the time necessary to generate invariants and build the initial abstraction. The field N_{refs} represents the number of refinements. The only case in which refinement was needed is the *Red-black delete* example, which was verified using the *Infeasible Elementary Loop* refinement heuristic (Section 3.4).

6 CONCLUSIONS

We proposed a new generic termination-analysis framework. In this framework, infinite runs of a program are abstracted by Büchi automata. This abstraction is then

intersected with a predefined automaton representing potentially infinite runs. In case of non-empty intersection, a counterexample is exhibited. If the counterexamples is proved to be spurious, the abstraction is refined. We instantiated the framework for programs manipulating tree-like data structures and we experimented with a prototype implementation, on top of the ARTMC invariant generator. Test cases include a number of classical algorithms that manipulate tree-like data structures.

Future work includes instantiation of the method for the class of programs handled by a tool called *Forester* [19] based on a tuples of tree automata. The encoding of complex data structures used in Forester is more flexible than that used in ARTMC and it would allow us to handle much bigger programs, as well as more complex and tricky pointer manipulations. Using the proposed method, we would also like to tackle the termination analysis for concurrent programs. Moreover, we would like to investigate methods for automated discovery of well-founded orderings on the complex data domains such as trees and graphs.

Acknowledgements

This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education, Youth, and Sports (project MSM 0021630528), the BUT FIT project FIT-S-12-1, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070., and the French National Research Agency (project VERIDIC ANR-09-SEGI-016 (2009-2013)).

REFERENCES

[1] Berdine, J.—Chawdhary, A.—Cook, B.—Distefano, D.—O'Hearn, P.: Variance Analyses from Invariance Analyses. In Proc. of POPL '07, ACM Press, 2007.

[2] Blondel, V. D.—Portier, N.: The Presence of a Zero in an Integer Linear Recurrent Sequence is NP–Hard to Decide. In Linear Algebra and Its Applications, Vol. 351–352, 2002, pp. 91–98.

[3] Bouajjani, A.—Bozga, M.—Habermehl, P.—Iosif, R.—Moro, P.—Vojnar, T.: Programs with Lists are Counter Automata. In Proc. of CAV '06, Vol. 4144 of LNCS, Springer, 2006.

[4] Bouajjani, A.—Habermehl, P.—Rogalewicz, A.—Vojnar, T.: ARTMC: Abstract Regular Tree Model Checking. `http://www.fit.vutbr.cz/research/groups/verifit/tools/artmc/`.

[5] Bouajjani, A.—Habermehl, P.—Rogalewicz, A.—Vojnar, T.: Abstract Regular Tree Model Checking. ENTCS, Vol. 149, 2006, pp. 37–48. A preliminary version was presented at Infinity '05.

[6] Bouajjani, A.—Habermehl, P.—Rogalewicz, A.—Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In Proc. of SAS '06, Vol. 4134 of LNCS, Springer, 2006.

[7] Bozga, M.—Iosif, R.—Lakhnech, Y.: Flat Parametric Counter Automata. In Proc. of ICALP '06, Vol. 4052 of LNCS, Springer, 2006.

[8] Bradley, A. R.—Manna, Z.—Sipma, H. B.: Termination of Polynomial Programs. In Proc. of VMCAI 2005, Vol. 3385 of LNCS, Springer, 2005.

[9] Bradley, A. R.—Manna, Z.—Sipma, H. B.: The Polyranking Principle. In Proc. of ICALP '05, Vol. 3580 of LNCS, Springer, 2005.

[10] Braverman, M.: Termination of Integer Linear Programs. In Proc. of CAV '06, Vol. 4144 of LNCS, Springer, 2006.

[11] Clarke, E. M.—Grumberg, O.—Jha, S.—Lu, Y.—Veith, H.: Counterexample-Guided Abstraction Refinement. In Proc. of CAV 2000, Vol. 1855 of LNCS, Springer, 2000.

[12] Colussi, L.—Marchiori, E.—Marchiori, M.: On Termination of Constraint Logic Programs. In Proc. of CP '95, Vol. 976 of LNCS, Springer, 1995.

[13] Comon, H.—Dauchet, M.—Gilleron, R.—Jacquemard, F.—Lugiez, D.—Tison, S.—Tommasi, M.: Tree Automata Techniques and Applications, 2005, `http://www.grappa.univ-lille3.fr/tata`.

[14] Cook, B.—Podelski, A.—Rybalchenko, A.: Abstraction Refinement for Termination. In Proc. of SAS '05, Vol. 3672 of LNCS, 2005.

[15] Cormen, T. H.—Leiserson, C. E.—Rivest, R. L.: Introduction to Algorithms. The MIT Press, 1990.

[16] Courcoubetis, C.—Vardi, M. Y.—Wolper, P.—Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In Proc. of CAV '90, Vol. 531 of LNCS, Springer, 1991.

[17] Distefano, D.—Berdine, J.—Cook, B.—O'Hearn, P. W.: Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In Proc. of CAV '06, Vol. 4144 of LNCS, Springer, 2006.

[18] Finkel, A.—Leroux, J.: How to Compose Presburger-Accelerations: Applications to Broadcast Protocols. In Proc. of FSTTCS '02, Vol. 2556 of LNCS, Springer, 2002.

[19] Habermehl, P.—L. Holík, Rogalewicz, A.—J. Šimáček—Vojnar, T.: Forest Automata for Verification of Heap Manipulation. In Proc. of CAV 2011, Vol. 6806 of LNCS, Springer Verlag, 2011.

[20] Habermehl, P.—Iosif, R.—Rogalewicz, A.—Vojnar, T.: Proving Termination of Tree Manipulating Programs. In Proc. of ATVA '07, Vol. 4762 of LNCS, Springer, 2007.

[21] Iosif, R.—Rogalewicz, A.: Automata-Based Termination Proofs. In Proc. of CIAA 2005, Vol. 5642 of LNCS, Springer, 2009.

[22] Khoussainov, B.—Nerode, A.: Automata Theory and Its Applications. Birkhauser Boston, 2001.

[23] Lahiri, S. K.—Qadeer, S.: Verifying Properties of Well-Founded Linked Lists. In Proc. of POPL '06, ACM Press, 2006.

[24] Lee, C. S.—Jones, N. D.—Ben-Amram, A. M.: The Size-Change Principle for Program Termination. In Proc. of POPL 2001, ACM Press, 2001.

[25] Loginov, A.—Reps, T. W.—Sagiv, M.: Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In Proc. of SAS '06, Vol. 4134 of LNCS, Springer, 2006.

[26] Podelski, A.—Rybalchenko, A.: Transition Invariants. In Proc. of LICS '04. IEEE, 2004.

[27] Rozenberg, G.—Salomaa, A. editors: Handbook of Formal Languages, Vol. 3: Beyond Words. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[28] Rybalchenko, A.: ARMC: Abstraction Refinement Model Checker. `http://www.mpi-inf.mpg.de/~rybal/armc/`.

[29] Vardi, M. Y.: The Büchi Complementation Saga. In Proc. of STACS '07, Vol. 4393 of LNCS, Springer, 2007.

Radu Iosif is a full-time researcher at Centre National de Recherche Scientifique (CNRS), VERIMAG Laboratory (Grenoble, France). He received his M. Sc. degree from the Politechnic University of Bucharest (Romania), and his Ph. D. degree from Politecnico di Torino (Italy), both in computer science. After completing his Ph. D., he spent 2 years as a research assistant at Kansas State University (USA). His main research interests are in logic, automata theory and program verification.

Adam Rogalewicz is an Assistant Professor at Brno University of Technology, Faculty of Information Technology and member of VeriFIT research group. He received his M. Sc. degree at Masaryk University Brno and his Ph. D. degree at Brno University of Technology, both in computer science. After completing his Ph. D., he worked as a post-doc at Centre National de Recherche Scientifique (CNRS), VERIMAG Laboratory.

Appendix B

Forest Automata for Verification of Heap Manipulation

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

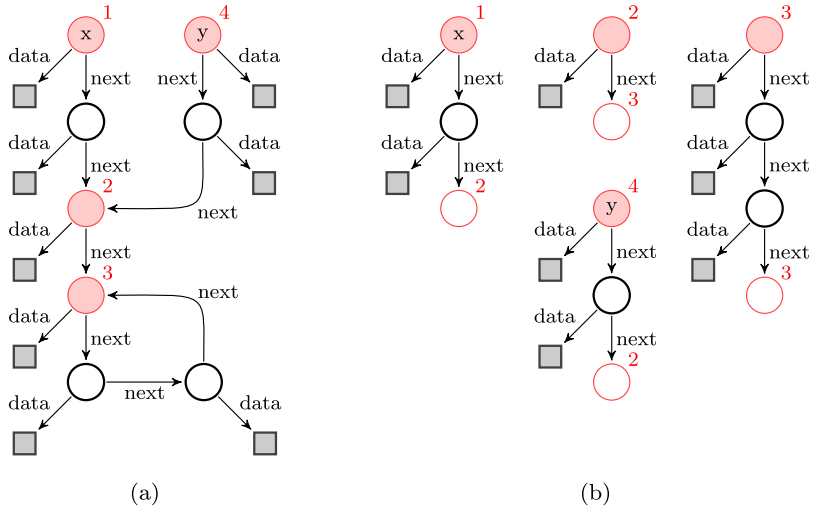


Fig. 1 (**a**) A heap graph with cut-points highlighted in red, (**b**) the canonical tree decomposition of the heap with `x` ordered before `y`

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.

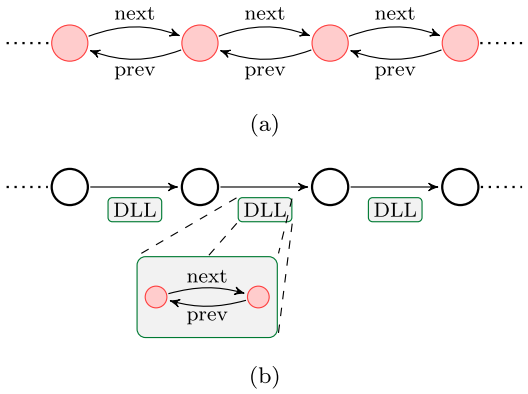


Fig. 2 (**a**) A part of a DLL, (**b**) a hierarchical encoding of the DLL

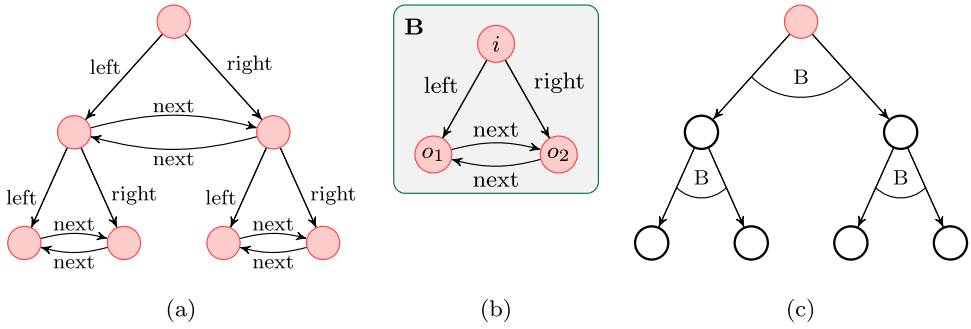


Fig. 3 (**a**) A tree with linked brother nodes, (**b**) a pattern that repeats in the structure and that is linked in such a way that all nodes in the structure are cut-points, (**c**) the tree with linked brother nodes represented using hyperedges labelled by the box *B*

ports *o*1, *o*2. 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 $\mathcal{A}^{\mathcal{F}} = (\Sigma \cup \{人_n^R\}, Q, \Delta, \{q^{top}\})$ where:

- $人_n^R \notin \Sigma$ is a fresh symbol with $\#(人_n^R) = n$,
- $q^{top} \notin \bigcup_{i=1}^{n} Q_i$ 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 $人_n^R(q_1, \ldots, q_n) \to q^{top}$ 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.

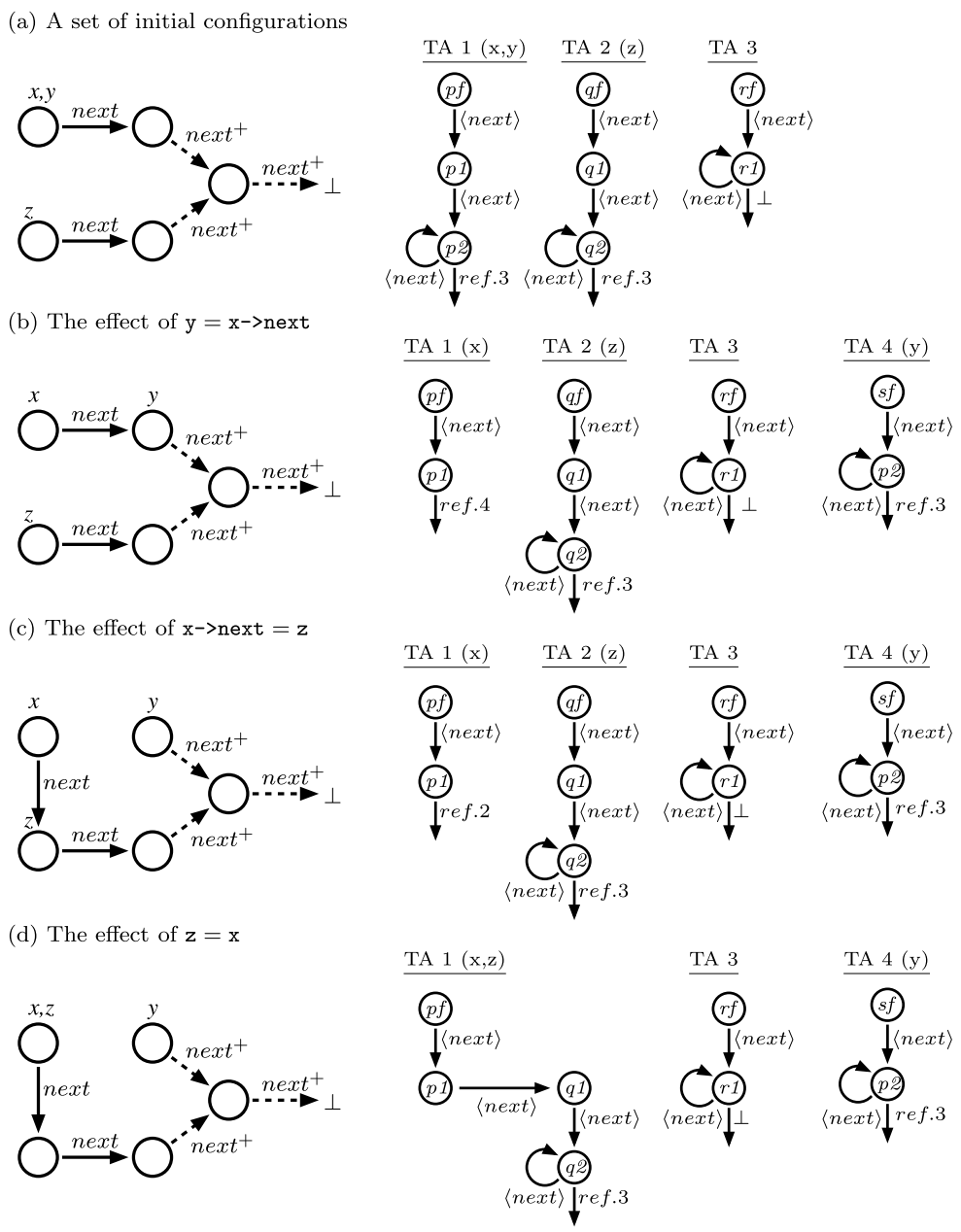


Fig. 4 A concrete (on the *left*) and symbolic execution (on the *right*) of statements `y = x->next`, `x->next = z`, and `z = x`. For the sake of simplicity, the presented FA are not strictly in their canonical form

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

Appendix C

Abstract Regular (Tree) Model Checking

Int J Softw Tools Technol Transfer (2012) 14:167–191
DOI 10.1007/s10009-011-0205-y

Abstract regular (tree) model checking

Ahmed Bouajjani · Peter Habermehl · Adam Rogalewicz · Tomáš Vojnar

Published online: 20 July 2011

Abstract Regular model checking is a generic technique for verification of infinite-state and/or parametrised systems which uses finite word automata or finite tree automata to finitely represent potentially infinite sets of reachable configurations of the systems being verified. The problems addressed by regular model checking are typically undecidable. In order to facilitate termination in as many cases as possible, acceleration is needed in the incremental computation of the set of reachable configurations in regular model checking. In this work, we describe how various incrementally refinable abstractions on finite (word and tree) automata can be used for this purpose. Moreover, the use of abstraction does not only increase chances of the technique to terminate, but it also significantly reduces the problem of an explosion in the number of states of the automata that are generated by regular model checking. We illustrate the efficiency of abstract regular (tree) model checking in verification of simple systems with various sources of infinity such as unbounded counters, queues, stacks, and parameters. We then show how abstract regular tree model checking can be used for verification of programs manipulating tree-like dynamic data structures. Even more complex data structures can be handled using a suitable tree-like encoding.

A. Bouajjani · P. Habermehl
LIAFA, Université Paris Diderot—Paris 7/CNRS,
Case 7014, 75205 Paris Cedex 13, France

A. Rogalewicz · T. Vojnar ( )
FIT, Brno University of Technology, Božetěchova 2,
61266 Brno, Czech Republic
e-mail: vojnar@fit.vutbr.cz

Keywords Formal verification · Infinite-state and parameterised systems · Programs with dynamic linked data structures · Regular model checking · Abstraction · Finite word and tree automata

1 Introduction

Model checking is nowadays widely accepted as a powerful technique for verification of finite-state systems. However, many real-life systems exhibit various aspects of infinity. In the case of discrete systems that we concentrate on in this paper, infinity can arise due to dealing with various kinds of *unbounded data structures* such as push-down stacks needed for dealing with recursive procedures, queues of waiting processes or messages, unrestricted counters (or integer variables), or dynamic linked data structures (such as lists or trees). A need to deal with infinite state spaces may also arise due to various kinds of *parameters* (such as the maximum value of some variable, the maximum length of a queue, or the number of processes in a system) when one wants to verify a given parametric system for any value of its parameters. In the last case, to be more precise, we are dealing with *infinite families* of systems which themselves may be finite-state or infinite-state. Nevertheless, the need to verify the system for any member of the family leads anyhow to infinite-state verification as the union of the state spaces of all the family members is infinite.

To deal with infinity in model checking, one can, e.g., try to identify sufficient finite bounds on the sources of infinity—the so called *cut-offs*, one can use various finite-range *abstractions*, or techniques of *automated induction* (for an overview of such techniques, see, e.g., [58]). Yet another approach is to use *symbolic model checking* based on a finite representation of infinite sets of states by means of logics,

automata, grammars, etc. Among successful symbolic verification methods, we have the so-called *regular* (*tree*) *model checking* R(T)MC, first mentioned in [36], on which we concentrate in this paper.

In R(T)MC, configurations of systems are encoded as words or trees over a finite alphabet whereas transitions are modelled as finite state transducers or, more generally, as regularity preserving relations on words or trees.[1] Finite (tree) automata can then naturally be used to represent and manipulate potentially infinite sets of configurations, allowing reachability properties to be checked by computing transitive closures of transducers [3,5,10,23,35] or images of automata by iteration of transducers [16,52]—depending on whether dealing with *reachability relations* or *reachability sets* is preferred. To facilitate termination of the computation, which is in general not guaranteed as the problem being solved is undecidable, various acceleration methods are usually used.

In this paper, we, in particular, concentrate on using *abstraction* as a means of acceleration. The description builds on our proposal [13,15] of combining R(T)MC with the *CEGAR* loop [21]. Instead of precise acceleration techniques, we use abstract fixpoint computations in some *finite* domain of automata. The abstract fixpoint computations always terminate and provide overapproximations of the reachability sets (relations). To achieve this, we define techniques that systematically map any automaton M to an automaton M' from some finite domain such that M' recognises a superset of the language of M. For the case that the computed overapproximation is too coarse and a spurious counterexample is detected, we provide effective techniques allowing the abstraction to be refined such that the new abstract computation does not encounter the same counterexample.

Both for the word and tree cases, we discuss two general purpose classes of techniques for abstracting automata.[2] They take into account the structure of the automata and are based on collapsing their states according to some equivalence relation. The first one is inspired by *predicate abstraction* [30]. However, contrary to classical predicate abstraction, we associate predicates with states of automata representing sets of configurations rather than with the configurations themselves. An abstraction is defined by a set of regular *predicate languages* L_P. We consider a state q of an automaton M to "satisfy" a predicate language L_P if the intersection of L_P with the language $L(M, q)$ accepted from the state q is not empty. Then, two states are equivalent if they satisfy the same predicates. The second abstraction technique is then based on considering two automata states equivalent if their *languages of words up to a certain fixed length* (or *trees up to a certain fixed height*) are equal. For both of these two abstraction methods, we provide effective refinement techniques allowing us to discard spurious counterexamples.

All the aforementioned techniques have up to now been implemented in prototype tools and tested on various case studies. In particular, abstract regular *word* model checking was successfully applied for verification of parametric networks of processes, pushdown systems, counter automata, systems with queues, and programs with dynamic singly linked structures [12,15]. Abstract regular *tree* model checking was applied for verification of parametric networks of processes [13] and programs with generic dynamic-linked data structures [14]. In this paper, we briefly report on all these applications and describe the last mentioned application in more detail.

2 Preliminaries

2.1 Finite word automata and transducers

A (non-deterministic) *finite-state automaton* is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ a transition function, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of final states. The transition relation $\xrightarrow[M]{} \subseteq Q \times \Sigma^* \times Q$ of M is defined as the smallest relation satisfying: (1) $\forall q \in Q : q \xrightarrow[M]{\varepsilon} q$, (2) if $q' \in \delta(q, a)$, then $q \xrightarrow[M]{a} q'$, and (3) if $q \xrightarrow[M]{w} q'$ and $q' \xrightarrow[M]{a} q''$, then $q \xrightarrow[M]{wa} q''$ for $a \in \Sigma, w \in \Sigma^*$. We drop the subscript M if no confusion is possible. M is called *deterministic* iff $\forall q \in Q\ \forall a \in \Sigma : |\delta(q, a)| \leq 1$.

The *language* recognised by a finite-state automaton $M = (Q, \Sigma, \delta, q_0, F)$ from a state $q \in Q$ is defined by $L(M, q) = \{w \in \Sigma^* \mid \exists q_F \in F : q \xrightarrow[M]{w} q_F\}$. The language $L(M)$ of M is equal to $L(M, q_0)$. A set $L \subseteq \Sigma^*$ is a *regular set* iff there exists a finite-state automaton M such that $L = L(M)$. We also define the *backward language* $\overleftarrow{L}(M, q) = \{w \mid q_0 \xrightarrow[M]{w} q\}$ and the *forward/backward languages of words up to a certain length*: $L^{\leq n}(M, q) = \{w \in L(M, q) \mid |w| \leq n\}$ and similarly $\overleftarrow{L}^{\leq n}(M, q)$. We define the *forward/backward trace languages* of states $T(M, q) = \{w \in \Sigma^* \mid \exists w' \in \Sigma^* : ww' \in L(M, q)\}$ and similarly $\overleftarrow{T}(M, q)$. Finally, we define accordingly forward/backward trace languages $T^{\leq n}(M, q)$ and $\overleftarrow{T}^{\leq n}(M, q)$ of *traces up to a certain length*.

Given a finite-state automaton $M = (Q, \Sigma, \delta, q_0, F)$ and an equivalence relation $\sim$ on its set of states Q, $M_{/\sim}$ denotes the *quotient automaton* of M wrt. $\sim$, $M_{/\sim} =$

[1] Such relations can be expressed, e.g., as special operations on automata.

[2] In [12], some specialised abstractions optimised for verification of list manipulating programs are proposed. These abstractions are, however, beyond the scope of this paper.

$(Q_{/\sim}, \Sigma, \delta_{/\sim}, [q_0]_{/\sim}, F_{/\sim})$ where $Q_{/\sim}$ and $F_{/\sim}$ are the partitions of Q and F wrt. $\sim$, respectively, $[q_0]_{/\sim}$ is the equivalence class of Q wrt. $\sim$ containing q_0, and $\delta_{/\sim}$ is defined s.t. $[q_1]_{/\sim} \xrightarrow[M_{/\sim}]{a} [q_2]_{/\sim}$ for $[q_1]_{/\sim}, [q_2]_{/\sim} \in Q_{/\sim}$, $a \in \Sigma$ iff $q_1' \xrightarrow[M]{a} q_2'$ for some $q_1' \in [q_1]_{/\sim}, q_2' \in [q_2]_{/\sim}$.

A *finite-state transducer* over Σ is a 5-tuple $\tau = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite input/output alphabet, $\delta : Q \times \Sigma_\varepsilon \times \Sigma_\varepsilon \to 2^Q$ a transition function, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of final states. A finite-state transducer is called a *length-preserving transducer* if its transitions do not contain ε. The transition relation $\xrightarrow[\tau]{} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ is defined as the smallest relation satisfying (1) $q \xrightarrow[\tau]{\varepsilon/\varepsilon} q$ for every $q \in Q$, (2) if $q' \in \delta(q, a, b)$, then $q \xrightarrow[\tau]{a/b} q'$, and (3) if $q \xrightarrow[\tau]{w/u} q'$ and $q' \xrightarrow[\tau]{a/b} q''$, then $q \xrightarrow[\tau]{wa/ub} q''$ for $a, b \in \Sigma_\varepsilon$, $w, u \in \Sigma^*$. The subscript τ will again be dropped if no confusion is possible. A finite-state transducer $\tau = (Q, \Sigma, \delta, q_0, F)$ defines the *relation* $\varrho_\tau = \{(w, u) \in \Sigma^* \times \Sigma^* \mid \exists q_F \in F : q_0 \xrightarrow[\tau]{w/u} q_F\}$.

A relation $\varrho \subseteq \Sigma^* \times \Sigma^*$ is a *regular relation* iff there exists a finite-state transducer τ such that $\varrho = \varrho_\tau$. For a set $L \subseteq \Sigma^*$ and a relation $\varrho \subseteq \Sigma^* \times \Sigma^*$, we denote by $\varrho(L)$ the set $\{w \in \Sigma^* \mid \exists w' \in L : (w', w) \in \varrho\}$. A relation $\varrho \subseteq \Sigma^* \times \Sigma^*$ is called *regularity preserving* iff $\varrho(L)$ is regular for any regular set $L \subseteq \Sigma^*$. Note that not all regularity preserving relations are regular—as an example of a regularity preserving, non-regular relation, one can take, e.g., the relation $\{(w, w^R) \mid w \in \Sigma^*\}$, for w^R being the reversal of w, $|\Sigma| > 1$.

2.2 Finite tree automata and transducers

A finite *alphabet* Σ is *ranked* if there exists a *rank* function $\# : \Sigma \to \mathbb{N}$. For each $k \in \mathbb{N}$, $\Sigma_k \subseteq \Sigma$ is the set of all symbols with rank k. Symbols of Σ_0 are called *constants*. Let χ be a denumerable set of symbols called *variables*. $T_\Sigma[\chi]$ denotes the set of *terms* over Σ and χ. The set $T_\Sigma[\emptyset]$ is denoted by T_Σ, and its elements are called *ground terms*. A term t from $T_\Sigma[\chi]$ is called *linear* if each variable occurs at most once in t.

A finite ordered *tree* t over a set of labels L is a mapping $t : \mathcal{P}os(t) \to L$ where $\mathcal{P}os(t) \subseteq \mathbb{N}^*$ is a finite, prefix-closed set of *positions* in the tree satisfying (1) for all $p \in \mathcal{P}os(t)$, if $t(p) \in \Sigma_n$ with $n \geq 1$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \{1, \ldots, n\}$ and (2) for all $p \in \mathcal{P}os(t)$, if $t(p) \in \Sigma_0 \cup \chi$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \emptyset$. A term $t \in T_\Sigma[\chi]$ can naturally be also viewed as a tree whose leaves are labelled by constants and variables, and each node with k sons is labelled by a symbol from Σ_k [22]. Therefore, below, we sometimes exchange terms and trees. We denote by $\mathcal{NL}Pos(t) = \{p \in \mathcal{P}os(t) \mid \exists i \in \mathbb{N} : pi \in \mathcal{P}os(t)\}$ the set of *non-leaf* positions.

A *bottom-up tree automaton* over a ranked alphabet Σ is a tuple $A = (Q, \Sigma, F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, and δ is a set of transitions of the following types: (i) $f(q_1, \ldots, q_n) \to_\delta q$, (ii) $a \to_\delta q$, and (iii) $q \to_\delta q'$ where $a \in \Sigma_0$, $f \in \Sigma_n$, and $q, q', q_1, \ldots, q_n \in Q$. Below, we denote bottom-up tree automata simply as tree automata.

Let t be a ground term. A run of a tree automaton A on t is defined as follows: First, leaves are labelled with states. If a leaf is a symbol $a \in \Sigma_0$ and there is a rule $a \to_\delta q \in \delta$, the leaf is labelled by q. An internal node $f \in \Sigma_k$ is labelled by q if there exists a rule $f(q_1, q_2, \ldots, q_k) \to_\delta q \in \delta$ and the first son of the node has the state label q_1, the second one $q_2, \ldots$, and the last one q_k. Rules of the type $q \to_\delta q'$ are called *ε-steps* and allow us to change a state label from q to q'. If the top symbol is labelled with a state from the set of final states F, the term t is accepted by the automaton A.

A set of ground terms accepted by a tree automaton A is called a *regular tree language* and is denoted by $L(A)$. Let $A = (Q, \Sigma, F, \delta)$ be a tree automaton and $q \in Q$ a state; then we define the *language of the state q*—$L(A, q)$—as the set of ground terms accepted by the tree automaton $A_q = (Q, \Sigma, \{q\}, \delta)$. The language $L^{\leq n}(A, q)$ is defined to be the set $\{t \in L(A, q) \mid height(t) \leq n\}$.

A *bottom-up tree transducer* is a tuple $\tau = (Q, \Sigma, \Sigma', F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is an input ranked alphabet, Σ' is an output ranked alphabet, and δ is a set of transition rules of the following types: (i) $f(q_1(x_1), \ldots, q_n(x_n)) \to_\delta q(u)$, $u \in T_{\Sigma'}[\{x_1, \ldots, x_n\}]$, (ii) $q(x) \to_\delta q'(u)$, $u \in T_{\Sigma'}[\{x\}]$, and (iii) $a \to_\delta q(u)$, $u \in T_{\Sigma'}$ where $a \in \Sigma_0$, $f \in \Sigma_n$, $x, x_1, \ldots, x_n \in \chi$, and $q, q', q_1, \ldots, q_n \in Q$. In the following, we call a bottom-up tree transducer simply a tree transducer. We always use tree transducers with $\Sigma = \Sigma'$.

A run of a tree transducer τ on a ground term t is similar to a run of a tree automaton on this term. First, rules of type (iii) are used. If a leaf is labelled by a symbol a and there is a rule $a \to_\delta q(u) \in \delta$, the leaf is replaced by the term u and labelled by the state q. If a node is labelled by a symbol f, there is a rule $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \to_\delta q(u) \in \delta$, the first subtree of the node has the state label q_1, the second one $q_2, \ldots$, and the last one q_n, then the symbol f and all subtrees of the given node are replaced according to the right-hand side of the rule with the variables $x_1, \ldots, x_n$ substituted by the corresponding left-hand-side subtrees. The state label q is assigned to the new tree. Rules of type *(ii)* are called *ε-steps*. They allow us to replace a q-state-labelled tree by the right-hand side of the rule and assign the state label q' to this new tree with the variable x in the rule substituted by the original tree. A run of a transducer is successful if the root of a tree is processed and is labelled by a state from F.

A tree transducer is *linear* if all right-hand sides of its rules are linear (no variable occurs more than once). The class of linear bottom-up tree transducers is closed under composition. A tree transducer is called *structure-preserving* (or a *relabelling*) if it does not modify the structure of input trees and just changes the labels of their nodes. A transducer τ defines the relation $\varrho_\tau = \{(t, t') \in T_\Sigma \times T_\Sigma \mid t \rightarrow_\delta^* q(t')$ for some $q \in F\}$. For a set $L \subseteq T_\Sigma$ and a relation $\varrho \subseteq T_\Sigma \times T_\Sigma$, we denote $\varrho(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w', w) \in \varrho\}$ and $\varrho^{-1}(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w, w') \in \varrho\}$. If τ is a linear tree transducer and L is a regular tree language, then the sets $\varrho_\tau(L)$ and $\varrho_\tau^{-1}(L)$ are regular and effectively constructible [22,27]. Finally, the notions of *regular tree relations* and *regularity preserving tree relations* can be introduced analogously to the word case.

3 Regular model checking

3.1 The basic idea

As we have already mentioned in the introduction, the basic idea behind regular model checking is to encode particular configurations of the considered systems as *words over a suitable finite alphabet* and to represent infinite, but regular, sets of such configurations by *finite-state automata*. Transitions between the configurations, constituting the one-step transition relation of the given system, are then encoded using (one or more) *finite-state transducers*, or, more generally, using (one or more) regularity preserving relations expressed, e.g., by specialised automata operations.[3] In this section, we, for simplicity, concentrate on using a single transducer encoding the one-step transition relation of a given system.

Before going into more technical details of regular model checking, we present a simple illustrating example from the area of verification of parametric networks of processes with a linear topology. When dealing with such systems, each letter in a word representing a configuration will typically model the state of a single process, and the length of the word will correspond to the number of processes in the given instance of the system. Let us in particular consider a very simple token passing protocol. We have an arbitrary, but finite number of processes arranged into a linear network. Each process either does not have a token and is waiting for a token to arrive from its left neighbour, or it has a token and then it can pass it to its right neighbour. We suppose that initially there is only one token which is owned by the left-most process. To encode

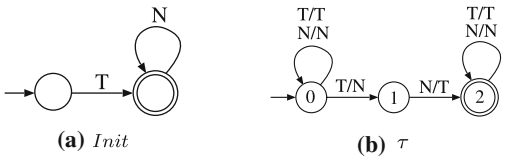


Fig. 1 A model of a simple token passing protocol: (**a**) an automaton $Init$ encoding the initial set of configurations $I = L(Init)$, (**b**) a transducer τ encoding the 1-step transition relation $\varrho = \varrho_\tau$

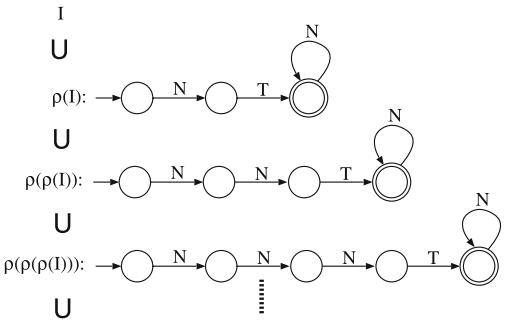


Fig. 2 Divergence of the non-accelerated reachability set computation $\varrho^*(I) = I \cup \varrho(I) \cup \varrho(\varrho(I)) \cup \ldots$ for the protocol from Fig. 1

the state of each process in our protocol, we suffice with the alphabet $\Sigma = \{N, T\}$ where N means that the process does not have a token, whereas T means the process has a token. Then, the set I of all possible initial configurations can be encoded by the automaton $Init$ shown in Fig. 1a and the single-step transition relation by the transducer τ in Fig. 1b.

Once we have a transducer encoding the single-step transition relation ϱ of the system and an automaton encoding its set of initial configurations I, there are two basic strategies we can follow. We can either try to directly compute the set of all reachable configurations $\varrho^*(I)$, or the reachability relation ϱ^* of the system. The set $\varrho^*(I)$ can be obtained by repeatedly applying the single-step transition relation ϱ on the set of the so far reached states and by taking the union of all such sets, i.e., $\varrho^*(I) = I \cup \varrho(I) \cup \varrho(\varrho(I)) \cup \ldots$. On the other hand, the reachability relation ϱ^* can be obtained by repeatedly composing ϱ with the so far computed reachability relation and by taking the union of all such relations, i.e., $\varrho^* = \iota \cup \varrho \cup (\varrho \circ \varrho) \cup (\varrho \circ \varrho \circ \varrho) \cup \ldots$ where ι is the identity relation.

The problem is that in the context of parameterised and infinite-state systems, if we try to compute the above infinite unions using a straightforward fixpoint computation, the computation will usually not terminate. We can illustrate this even on our simple token passing protocol. In Fig. 2, we give the first members of the sequence I, $\varrho(I)$, $\varrho(\varrho(I))$, $\varrho(\varrho(\varrho(I)))$, ..., which clearly show that a fixpoint will never be reached (the token can be at the begin-

[3] Several transducers/regularity preserving relations may always be united into a single transducer/regularity preserving relation, respectively. Dealing with one complex or more simple transducers or regularity preserving relations may, however, differ in efficiency in different scenarios.

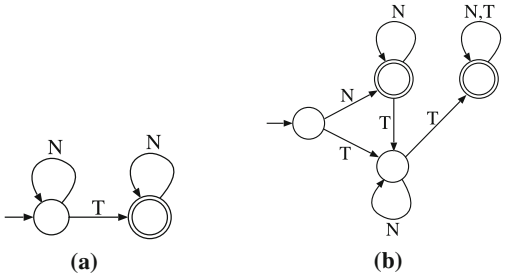


Fig. 3 The simple token passing protocol—automata encoding the set of (**a**) reachable and (**b**) bad configurations

ning, one step to the right, two steps to the right, three steps to the right, etc.).

In order to make the computation of $\varrho^*(I)$ or ϱ^* terminate at least in many practical cases, we need some kind of *acceleration* of the computation which will allow us to obtain the result of an infinite number of the described computation steps at once (i.e., in some sense, to "jump" to the fixpoint). We can, e.g., notice that in our example, the token is moving step-by-step to the right, and we can accelerate the fixpoint computation by allowing the token to move arbitrarily far to the right in one step. If we use such an acceleration, we will immediately reach the fixpoint shown in Fig. 3a, which represents the set of all reachable configurations of our protocol. In the literature, several different approaches to a systematic acceleration of fixpoint computations in regular model checking have been proposed. We will very briefly review them in Sect. 3.3.

3.2 Verification by regular model checking

It is well known that checking of *safety properties* can be reduced to checking that no "bad" states are reachable in the given system. If the set of bad states, for which we want to check that they are not reachable in the given system, can be expressed as a regular set B, we may simply compute the reachability set $\varrho^*(I)$ and check that $\varrho^*(I) \cap B = \emptyset$.

For instance, in our simple token passing protocol, we can consider as bad the situation when there is no token in the system or when there appear two or more tokens. The set of such bad states is encoded by the finite-state automaton in Fig. 3b, and it is clear that its intersection with the set of reachable states from Fig. 3a is empty, and thus the system is safe in the given sense.

Checking of *liveness properties* within regular model checking is considerably more difficult. In the world of finite-state systems, it is known that liveness can be reduced to the repeated reachability problem. A similar approach can be taken in the context of regular model checking when the studied systems are modelled by *length-preserving transition relations*, which is typical, e.g., for parameterised networks of processes. In such cases, clearly, the only way how a system can loop is to repeatedly go through some configuration. In a similar way as above, we can then instrument the system by a Büchi automaton[4] (or automata) encoding the undesirable behaviours, and check, e.g., that $\varrho^*(I) \cap A \cap domain(\varrho^+ \cap \iota) = \emptyset$. Here, A is the set of accepting configurations, ι is the identity relation, and $domain$ is the projection of a relation onto its domain.

Note that in the above described computation, we need to compute not only the reachability set, but also the reachability relation. Nevertheless, this step may be avoided by guessing when an accepting cycle begins, doubling every letter in the given configuration word, then continuing the computation only on the even letters and detecting a closure of the loop by looking for a situation when all the even letters correspond to the odd ones—we have practically tested this technique in some of the experiments presented in Sect. 4.5 (and it was studied more deeply in [49]).

A systematic framework for modelling parameterised networks of processes as well as specifying their properties to be checked via regular model checking (including liveness properties) has been proposed in [2,3]. The framework uses as a modelling as well as a specification language LTL(MSO) that is a combination of the linear time temporal logic LTL for expressing temporal relations and the monadic second-order logic on words for expressing properties on configuration words. (The MSO part is used for specifying, e.g., that every process in a configuration has to satisfy some condition, or that in the configuration there must exist a process for which some condition holds, and so on.) The work also proposes an automatic translation of the models as well as properties to be checked over them into an automata framework suitable for regular model checking. A computation of the reachability relation is then used for the actual verification.

Finally, checking liveness properties for systems modelled using *non-length-preserving transition relations* is even more complex than checking liveness in the length-preserving case. This is because a non-length-preserving system may exhibit infinite behaviours infinitely going through an accepting state of the monitoring Büchi automaton even when it does not loop at all—it suffices to imagine a system with a queue that keeps growing beyond every bound. For such cases, [17] has proposed an approach based on using regular model checking for automatically computing the greatest simulation relation on the reachable configurations which is compatible with the property being tracked. Then, instead of checking that an accepting configuration can be reached that is reachable from itself too, one checks that an accepting

[4] Büchi automata are finite automata that accept infinite words by infinitely looping through some of their accepting states (for a formal definition and the associated theory see, e.g., [43]).

configuration c_1 is reachable from which an accepting configuration c_2 simulating c_1 (i.e., allowing at least the same behaviours from the point of view of the tracked property) is reachable. An alternative approach based on learning fixpoints of specially proposed modalities from their generated samples using language inference algorithms has then been proposed in [56].

3.3 Acceleration in regular model checking

Acceleration methods designed for regular model checking include acceleration schemes [44], quotienting [5], extrapolation [16,39], inference of regular languages [29,33], and abstraction of automata [15]. The use of abstraction is described in detail in Sect. 4. A short description of the other methods is given below.

The use of *acceleration schemes* has been proposed in [44]. Acceleration schemes allow one to derive (from the original transitions of a system) meta-transitions encoding the effect of firing some of the original transitions an arbitrary number of times. The work [44] has provided three particular schemes for which it is experimentally checked that they suffice for verification of many cases of parameterised networks of processes. In particular, the following schemes are considered: (1) *local acceleration* allowing an arbitrary number of successive transitions of a single process to be fired at once, (2) *global acceleration of unary transitions* allowing any number of processes to fire a certain transition in a sequential order within one accelerated step, and (3) *global acceleration of binary transitions* allowing any number of processes to fire in a sequential order two consecutive transitions each—and thus communicate with both of its neighbours—in one atomic step (this way, e.g., a token in a token passing protocol can "jump" any number of positions ahead in one accelerated step). This method has been implemented in the TLV[P] tool [50].

The *quotienting technique* has been elaborated in the works [4,5,16,23,35,41,42]. Let $\tau = (Q, \Sigma, \delta, q_0, F)$ be a length-preserving transducer encoding the single-step transition relation ϱ of a system being examined. The basic idea of the quotienting technique stems from viewing the result of an arbitrary number of compositions of ϱ encoded by τ as an infinite-state "history" transducer $\tau_{hist} = (Q^+, \Sigma, \delta_{hist}, \{q_0\}^+, F^+)$ whose states[5] reflect the history of their creation in terms of which states of τ have been passed at a particular position in a word in the first, second, and further transductions. Therefore, δ_{hist} is defined such that $q_1q_2\ldots q_n \xrightarrow[\tau_{hist}]{a/a'} q'_1q'_2\ldots q'_n$ for some $n \geq 1$ iff there exist $a_1, a_2, \ldots, a_{n+1} \in \Sigma$ such that $a = a_1$, $a' = a_{n+1}$, and $\forall i \in \{1,\ldots,n\} : q_i \xrightarrow[\tau]{a_i/a_{i+1}} q'_i$. Intuitively, this means that $q_1q_2\ldots q_n \xrightarrow[\tau_{hist}]{a/a'} q'_1q'_2\ldots q'_n$ represents the composition of the $q_i \xrightarrow[\tau]{a_i/a_{i+1}} q'_i$ transductions for $i = 1,\ldots,n$. Clearly, τ_{hist} encodes the reachability relation ϱ^+. Of course, the history transducer τ_{hist} is of no practical use as it is infinite-state. The idea is to come up with some *column equivalence* $\simeq$ on its states—i.e., on sequences (or, in the original terminology, columns) of states of the original transducer τ—such that the *quotient transducer* $\tau_{hist/\simeq}$ is (1) finite-state as often as possible, and at the same time, (2) describes exactly the same relation as τ_{hist}. Suitable column equivalences have been proposed along with ways on how to build the quotient transducer incrementally (e.g., by gradually adding new transitions obtained by composing transitions as described above while also gradually quotienting the automaton—obviously, one cannot construct a history transducer and only then quotient it).

The *extrapolation* (or *widening*) approach to regular model checking [10,16] is based on comparing successive elements of the sequence I, $\varrho(I)$, $\varrho(\varrho(I)), \ldots$, trying to find some repeated growth pattern in it, and adding an arbitrary number of its occurrences into the reachability set. In particular, following [16], let $L \subseteq \Sigma^*$ be a so far computed reachability set and $\varrho \subseteq \Sigma^* \times \Sigma^*$ a regular one-step transition relation. One can check whether there are regular sets L_1, L_2, and Δ satisfying the following two conditions: (C1) $L = L_1.L_2$ and $\varrho(L) = L_1.\Delta.L_2$ and (C2) $L_1.\Delta^*.L_2 = \varrho(L_1.\Delta^*.L_2) \cup L$. If the conditions hold, $L_1.\Delta^*.L_2$ is added to the so far computed reachability set. Intuitively, C1 means that the effect of applying ϱ is to add Δ between L_1 and L_2. C2 then ensures that $\varrho^*(L) \subseteq L_1.\Delta^*.L_2$, and so we add at least all the configurations reachable from L by iterating ϱ. Note that the exactness of the acceleration—i.e., whether $L_1.\Delta^*.L_2 \subseteq \varrho^*(L)$ holds too—is not guaranteed in general. However, [16] gives a sufficient condition on ϱ under which C1 and C2 lead to an exact acceleration. This condition in particular requires ϱ to be *well-founded*, i.e., not allowing any word to have an infinite number of predecessors wrt. ϱ. There is also a syntactic criterion for the so-called *simple rewriting relations* that are guaranteed to satisfy the above condition and that seem to appear quite often in practice.

Regular model checking based on *inference of regular languages* was studied in [33] extending [29]. Here, an important observation is that, for an infinite-state system whose behaviour is described by a length-preserving transducer τ, a set containing all reachable words up to the given length can be computed by a simple iterative application of τ on the set of initial configurations. These configurations are taken as a sample. Then some language inference algorithm may be applied to learn the whole reachable set (or its overapproximation) from this sample. As shown in [33], termination of

[5] We allow here a set of initial states.

the method is guaranteed whenever the set of all reachable configurations is regular. This is not the case for other acceleration methods. In [54–57], similar results were proved, covering even omega-regular model checking and checking of branching-time properties.

4 Abstract regular model checking

Apart from the need to accelerate the reachability computation to make it terminate in as many practical scenarios as possible, another crucial problem to be faced in regular model checking is the *state space explosion in automata* (*transducer*) *representations* of the sets of configurations (or reachability relations) being examined. One of the sources of this problem is related to the nature of the previously mentioned regular model checking techniques. Typically, these techniques try to calculate the *exact* reachability sets (or relations) independently of the property being verified. However, it is often enough to only compute an overapproximation of the reachability set (or relation) precise enough just to verify the given property of interest. Indeed, this is the way how large (or infinite) state spaces are often being successfully handled outside the domain of regular model checking using the so-called *abstract-check-refine* paradigm often implemented in the form of some *counterexample-guided abstraction refinement* (CEGAR) loop [8,21,24,30,34,48].

Inspired by the above, we have proposed in [15] a new approach to regular model checking which is based on the abstract-check-refine paradigm. Instead of a precise acceleration, we use abstract fixpoint computations in some *finite* domain of automata. As we have already briefly mentioned in the introduction, the abstract fixpoint computations always terminate and provide overapproximations of the reachability sets (relations). To achieve this, we define techniques that systematically map any automaton M to an automaton M' from some finite domain such that M' recognises a superset of the language of M.

The abstraction techniques we discuss below take into account the structure of automata and are based on collapsing their states according to some equivalence relation. The first one is inspired by predicate abstraction. We consider a state q of an automaton M to "satisfy" a predicate language L_P if the intersection of L_P with the language $L(M, q)$ accepted from the state q is not empty. Subsequently, two states are equivalent if they satisfy the same predicates. The second abstraction technique is then based on considering two automata states equivalent if their *languages of words up to a certain fixed length* are equal. For both of these two abstraction methods, we provide effective refinement techniques allowing us to discard spurious counterexamples.

We also introduce several natural alternatives to the above basic approaches, based on backward and/or trace languages

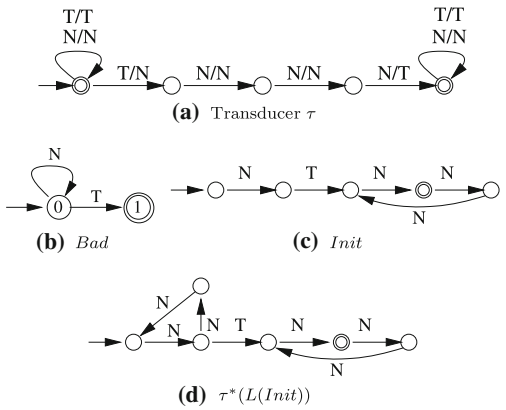


Fig. 4 A transducer τ modelling a modified token passing protocol and automata describing the initial, bad, and reachable configurations of the system

of states of automata. For them, it is not always possible to guarantee the exclusion of a spurious counterexample, but according to our experience, they still provide good practical results.

All of our techniques can be applied to dealing with reachability sets (obtained by iterating length-preserving or even general transducers) as well as length-preserving reachability relations.

4.1 A running example and some basic assumptions

As a simple running example capable of illustrating the different techniques of abstract regular model checking (ARMC) that we discuss here, we consider a slight modification of the token passing protocol from Fig. 1. The modification consists in that each process can pass the token to its *third* right neighbour (instead of its direct right neighbour). The one-step transition relation of the system is encoded by the transducer τ in Fig. 4a. The transducer includes the identity relation too. In the initial configurations described by the automaton $Init$ from Fig. 4c, the second process has the token, and the number of processes is divisible by three. We want to show that it is not possible to reach any configuration where the last process has the token. This set is described by the automaton Bad from Fig. 4b.

Note that in the following, in order to shorten the descriptions, we *identify a transducer and the relation it represents* and write $\tau(L)$ instead of $\varrho_\tau(L)$. Let $\iota \subseteq \Sigma^* \times \Sigma^*$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations (transducers) $\tau^0 = \iota$, $\tau^{i+1} = \tau \circ \tau^i$, and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. As in our running example, we suppose

$\iota \subseteq \tau$ for the rest of the section meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

For our running example, $\tau^*(L(Init))$ is depicted in Fig. 4d, and the property of interest clearly holds. However, in general, $\tau^*(L(Init))$ is neither guaranteed to be regular nor computable. In the following, the verification task is thus to find a regular overapproximation $L \supseteq \tau^*(L(Init))$ such that $L \cap L(Bad) = \emptyset$.

4.2 The method of abstract regular model checking

We now describe the general principle of abstract regular model checking (ARMC) using a generic framework for automata abstraction based on collapsing states of the automata. This framework is then instantiated in several concrete ways in the following two sections. For simplicity, we restrict to the case where the one-step transition relation of the system at hand is given by a single transducer (the more general cases being analogical). Moreover, we concentrate on the use of ARMC for computing reachability sets only. However, ARMC can be applied for dealing with reachability relations too—though in the context of length-preserving transducers only.[6]

4.2.1 The basic framework of automata abstraction

Let Σ be a finite alphabet and $\mathbb{M}_\Sigma$ the set of all finite automata over Σ. By an *automata abstraction function* α, we understand a function that maps every automaton M over Σ to an automaton $\alpha(M)$ whose language is an overapproximation of the one of M. To be more precise, for some abstract domain of automata $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$, α is a mapping $\mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ such that $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. We call α *finitary* iff its range $\mathbb{A}_\Sigma$ is finite.

Working conveniently on the level of automata, given a transition relation expressed as a transducer τ over Σ and an automata abstraction function α, we introduce the *abstract transition function* τ_α as follows: For each automaton $M \in \mathbb{M}_\Sigma$, $\tau_\alpha(M) = \alpha(\hat{\tau}(M))$ where $\hat{\tau}(M)$ is the minimal deterministic automaton of $\tau(L(M))$.[7] Now, we can iteratively compute the sequence $(\tau^i_\alpha(M))_{i\geq 0}$. Since we suppose $\iota \subseteq \tau$, it is clear that if α is finitary, there exists $k \geq 0$ such that $\tau^{k+1}_\alpha(M) = \tau^k_\alpha(M)$. The definition of α implies $L(\tau^k_\alpha(M)) \supseteq \tau^*(L(M))$. This means that in a finite number of steps, we can compute an overapproximation of the reachability set $\tau^*(L(M))$.

[6] Indeed, length-preserving transducers over an alphabet Σ can be seen as finite-state automata over $\Sigma \times \Sigma$.

[7] A generalisation of ARMC to dealing with nondeterministic automata is possible—cf. [11].

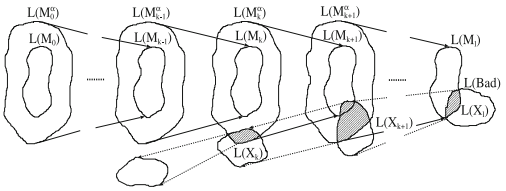


Fig. 5 A spurious counterexample in an abstract regular fixpoint computation

4.2.2 Refining automata abstractions

We call an automata abstraction function α' a *refinement* of α iff $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. A need to refine α arises when a situation depicted in Fig. 5 happens. Suppose we are checking whether no configuration from the set described by some automaton Bad is reachable from some given set of initial configurations described by an automaton M_0. We suppose $L(M_0) \cap L(Bad) = \emptyset$—otherwise, the property being checked is broken already by the initial configurations. Let $M^\alpha_0 = \alpha(M_0)$ and for each $i > 0$, $M_i = \hat{\tau}(M^\alpha_{i-1})$ and $M^\alpha_i = \alpha(M_i) = \tau_\alpha(M^\alpha_{i-1})$. There exist k and l $(0 \leq k < l)$ such that: (1) $\forall i : 0 \leq i < l : L(M_i) \cap L(Bad) = \emptyset$. (2) $L(M_l) \cap L(Bad) = L(X_l) \neq \emptyset$. (3) If we define X_i as the minimal deterministic automaton accepting $\tau^{-1}(L(X_{i+1})) \cap L(M^\alpha_i)$ for all i such that $0 \leq i < l$, then $\forall i : k < i < l : L(X_i) \cap L(M_i) \neq \emptyset$ and $L(X_k) \cap L(M_k) = \emptyset$ despite $L(X_k) \neq \emptyset$. Next, we see that either $k = 0$ or $L(X_{k-1}) = \emptyset$, and it is clear that we have encountered a *spurious counterexample*.

Note that when no l can be found such that $L(M_l) \cap L(Bad) \neq \emptyset$, the computation eventually reaches a fixpoint, and the property is proved to hold. On the other hand, if $L(X_0) \cap L(M_0) \neq \emptyset$, we have proved that the property is broken.

The *spurious counterexample may be eliminated* by refining α to α' such that for any automaton M whose language is disjoint with $L(X_k)$, the language of its α'-abstraction will not intersect $L(X_k)$ either. Then, the same faulty reachability computation (i.e., the same sequence of M_i and M^α_i) may not be repeated because we exclude the abstraction of M_k to M^α_k. Moreover, reachability of the bad configurations is in general excluded unless there is another reason for it than overapproximating by subsets of $L(X_k)$.

A slightly *weaker way of eliminating the spurious counterexample* consists in refining α to α' such that at least the language of the abstraction of M_k does not intersect with $L(X_k)$. In such a case, it is not excluded that some subset of $L(X_k)$ will again be used for an overapproximation somewhere, but we still exclude a repetition of exactly the same faulty computation. The obtained refinement can be coarser, which may lead to more refinements and a slower compu-

tation. On the other hand, the computation may terminate sooner due to quickly jumping to the fixpoint and use less memory due to working with less structured sets of configurations of the systems being verified—the abstraction is prevented from becoming unnecessarily precise in this case. For the latter reason, as illustrated later, one may sometimes successfully use even some more heuristic approaches that guarantee that the spurious counterexample will only eventually be excluded (i.e., after a certain number of refinements) or that do not guarantee the exclusion at all.

An obvious danger of using a heuristic approach that does not guarantee an exclusion of spurious counterexamples is that the computation may easily start looping. Notice, however, that even when we refine automata abstractions such that spurious counterexamples are always excluded, and the computation does not loop, we do not guarantee that it will eventually stop—we may keep refining forever. Indeed, the verification problem we are solving is undecidable in general.

4.2.3 Abstracting automata by collapsing their states

In the following two sections, we discuss several concrete automata abstraction functions. They are based on automata state equivalence schemas that define for each automaton from $\mathbb{M}_\Sigma$ an equivalence relation on its states. An automaton is then abstracted by collapsing all its states related by this equivalence. We suppose such an equivalence to reflect the fact that the future and/or history of the states to be collapsed is close enough, and the difference may be abstracted away.

Formally, an *automata state equivalence schema* $\mathbb{E}$ assigns an automata state equivalence $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ to each finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ over Σ. We define the *automata abstraction function* $\alpha_{\mathbb{E}}$ based on $\mathbb{E}$ s.t. $\forall M \in \mathbb{M}_\Sigma$: $\alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. We call $\mathbb{E}$ *finitary* iff $\alpha_{\mathbb{E}}$ is finitary. We *refine* $\alpha_{\mathbb{E}}$ by refining $\mathbb{E}$ such that more states are distinguished in at least some automata.

The automata state equivalence schemas presented below are then all based on one of the following two basic principles: (1) comparing states wrt. the intersections of their forward/backward languages with some *predicate languages* (represented by *predicate automata*) and (2) comparing states wrt. their forward/backward behaviours up to a certain *bounded length*.

4.3 State equivalences based on predicate languages

We start by introducing two automata state equivalence schemas defined wrt. a finite set of *predicate languages* represented by a set $\mathcal{P}$ of finite automata, which we denote as predicate automata. Namely, we introduce the schema $\mathbb{F}_\mathcal{P}$ based on forward languages of states and the schema $\mathbb{B}_\mathcal{P}$ based on backward languages. They compare two states of a given automaton according to the intersections of their forward/backward languages with the predicates.[8] Below, we first introduce the basic principles of the schemas and then add some implementation and optimisation notes.

4.3.1 The $\mathbb{F}_\mathcal{P}$ automata state equivalence schema

The automata state equivalence schema $\mathbb{F}_\mathcal{P}$ defines two states of a given automaton to be equivalent when their languages have a *nonempty intersection with the same predicates* of $\mathcal{P}$. Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, $\mathbb{F}_\mathcal{P}$ defines the state equivalence as the equivalence $\sim_M^\mathcal{P}$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^\mathcal{P} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(M, q_2) \neq \emptyset)$.

Clearly, as $\mathcal{P}$ is finite and there is only a finite number of subsets of $\mathcal{P}$ representing the predicates with which a given state has a nonempty intersection, $\mathbb{F}_\mathcal{P}$ is *finitary*.

For our example from Fig. 4, if we take as $\mathcal{P}$ the automata of the languages of the states of Bad, the automaton $Init$ from Fig. 4c is abstracted as follows: All states of $Init$ except the final one become equivalent since their languages have all empty intersections with the languages accepted from states 0 and 1 of Bad. Hence, when equivalent states are collapsed, we obtain the automaton in Fig. 6a, which after determinisation and minimisation gives the automaton in Fig. 6b. Then, the intersection of $\hat{\tau}(\alpha(Init))$ with the bad configurations—cf. Fig. 6d—is not empty, and we have to refine the abstraction.

The $\mathbb{F}_\mathcal{P}$ schema may be *refined by adding new predicates* into the current set of predicates $\mathcal{P}$. In particular, we can extend $\mathcal{P}$ by automata corresponding to the languages of all the states in X_k from Fig. 5. Theorem 1 shows that this prevents abstractions of languages disjoint with $L(X_k)$, such as—but not only—$L(M_k)$, from intersecting with $L(X_k)$. Consequently, as we have already explained, a repetition of the same faulty computation is excluded, and the set of bad configurations will not be reached unless there is another reason for this than overapproximating by subsets of $L(X_k)$.

Theorem 1 *Let $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ be any two finite automata and let $\mathcal{P}$ be a finite set of predicate automata such that $\forall q_X \in Q_X$: $\exists P \in \mathcal{P} : L(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X) = \emptyset$ too.*

Proof We prove the theorem by contradiction. Suppose $L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X) \neq \emptyset$. Let $w \in L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$.

[8] The use of intersection with predicate languages needs not be the only possible way of constructing some predicate language abstraction. Proposing a different abstraction based on predicate languages may be an interesting subject for further work. However, such an abstraction should come with some way of counterexample-guided refinement. This is not straightforward and we are currently not aware of any other refinable abstractions based on predicate languages than using the $\mathbb{F}_\mathcal{P}$ and $\mathbb{B}_\mathcal{P}$ schemas.

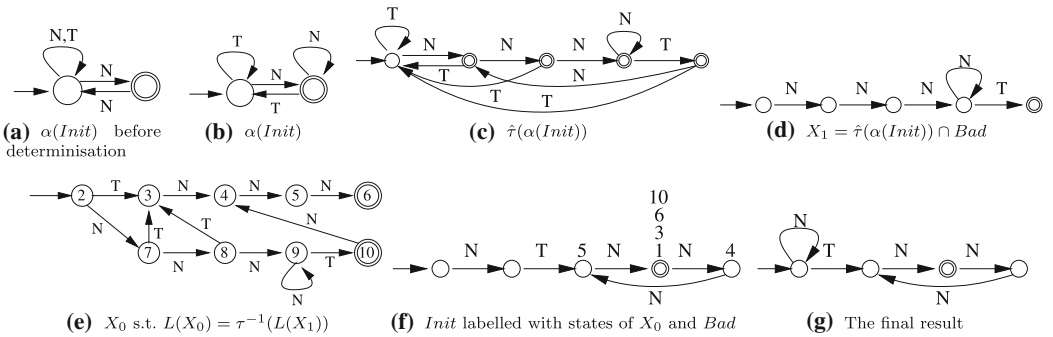


Fig. 6 An example using abstraction based on predicate languages

As w is accepted by $\alpha_{\mathbb{F}_\mathcal{P}}(M)$, M must accept it when we allow it to perform a certain number of "jumps" between states equal wrt. $\sim_M^\mathcal{P}$—after accepting a prefix of w and getting to some $q \in Q_M$, M is allowed to jump to any $q' \in Q_M$ such that $q \sim_M^\mathcal{P} q'$ and go on accepting from there (with or without further jumps).

Suppose that the minimum number of jumps needed to accept a word from $L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$ in M is i with $i > 0$, and let w' be such a word. Let the last jump within accepting w' in M be from some state $q_1 \in Q_M$ to some $q_2 \in Q_M$ such that $q_1 \sim_M^\mathcal{P} q_2$. Let $w' = w_1 w_2$ such that w_1 is read (possibly with jumps) just before the jump from q_1 to q_2. Clearly, $q_2 \xrightarrow[M]{w_2} q_3$ for some $q_3 \in F_M$. We know that X accepts w'. Suppose that after reading w_1, it is in some $q_X \in Q_X$. As $w_2 \in L(X, q_X)$ and $w_2 \in L(M, q_2)$, $L(M, q_2) \cap L(P) \neq \emptyset$ for the predicate(s) $P \in \mathcal{P}$ for which $L(P) = L(X, q_X)$. Moreover, as $q_1 \sim_M^\mathcal{P} q_2$, $L(M, q_1) \cap L(P) \neq \emptyset$ too. This implies there exists $w_2' \in L(P)$ such that $w_2' \in L(M, q_1)$ and $w_2' \in L(X, q_X)$. However, this means that $w_1 w_2' \in L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$ can be accepted in M with $i - 1$ jumps, which is a contradiction to the assumption of i being the minimum number of jumps needed. □

In our example, we refine the abstraction by extending $\mathcal{P}$ with the automata representing the languages of the states of X_0 from Fig. 6e. Figure 6f then indicates, for each state q of $Init$, the predicates corresponding to the states of Bad and X_0 whose languages have a non-empty intersection with the language of q. For example, the third state from the left of $Init$ is labelled by 5 because it accepts N which is also accepted by state 5 of X_0. The first two states of $Init$ are equivalent and are collapsed to obtain the automaton from Fig. 6g, which is a fixpoint showing that the property is verified. Notice that it is an overapproximation of the set of reachable configurations from Fig. 4d.

The price of refining $\mathbb{F}_\mathcal{P}$ by adding predicates for all the states in X_k may seem prohibitive, but fortunately this is not the case in practice. As described later on in Sect. 4.3.3, we do not have to treat all the new predicates separately. We exploit the fact that they come from one original automaton and share large parts of their structure. In fact, we can work just with the original automaton and each of its states may be considered an initial state of some predicate. This way, adding the original automaton as the only predicate and adding predicates for all of its states becomes roughly equal. Moreover, the refinement may be weakened by taking into account just some states of X_k as discussed later on.

4.3.2 The $\mathbb{B}_\mathcal{P}$ automata state equivalence schema

The $\mathbb{B}_\mathcal{P}$ automata state equivalence schema is an alternative of $\mathbb{F}_\mathcal{P}$ using *backward languages of states* rather than the forward ones. For an automaton $M = (Q, \Sigma, \delta, q_0, F)$, it defines the state equivalence as the equivalence $\overleftarrow{\sim}_M^\mathcal{P}$ such that $\forall q_1, q_2 \in Q : q_1 \overleftarrow{\sim}_M^\mathcal{P} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap \overleftarrow{L}(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap \overleftarrow{L}(M, q_2) \neq \emptyset)$.

Clearly, $\mathbb{B}_\mathcal{P}$ is *finitary* for the same reason as $\mathbb{F}_\mathcal{P}$. It may also be *refined* by extending $\mathcal{P}$ by automata corresponding to the languages of all the states in X_k from Fig. 5. Theorem 2 shows that the effect is the same as for $\mathbb{F}_\mathcal{P}$.

Theorem 2 *Let $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ be any two finite automata and let $\mathcal{P}$ be a finite set of predicate automata such that $\forall q_X \in Q_X : \exists P \in \mathcal{P} : \overleftarrow{L}(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{B}_\mathcal{P}}(M)) \cap L(X) = \emptyset$ too.*

Proof The theorem can be proved by contradiction in a similar way as Theorem 1. This time, as a consequence of working with backward languages of states, we do not deal with the last jump, but the first jump in accepting some $w' \in L(\alpha_{\mathbb{B}_\mathcal{P}}(M)) \cap L(X)$ in M. We do not look for a replacement w_2' of w_2 to be accepted from q_1 instead of q_2, but for a replacement w_1' of w_1 to be accepted before q_2 rather than before q_1. □

4.3.3 Optimising collapsing based on $\mathbb{F}_{\mathcal{P}}$ / $\mathbb{B}_{\mathcal{P}}$

The abstraction of an automaton M wrt. the automata state equivalence schema $\mathbb{F}_{\mathcal{P}}$ can be implemented by first labelling states of M by the states of predicate automata in $\mathcal{P}$ with whose languages they have a non-empty intersection and then collapsing the states of M that are labelled by the initial states of the same predicates (provided the sets of states of the predicate automata are disjoint). The labelling can be done in a way similar to constructing a backward synchronous product of M with the particular predicate automata: (1) $\forall P \in \mathcal{P}\ \forall q_F^P \in F_P\ \forall q_F^M \in F_M$: q_F^M is labelled by q_F^P, and (2) $\forall P \in \mathcal{P}\ \forall q_1^P, q_2^P \in Q_P\ \forall q_1^M, q_2^M \in Q_M$: if q_2^M is labelled by q_2^P, and there exists $a \in \Sigma$ such that $q_1^M \xrightarrow[\delta_M]{a} q_2^M$ and $q_1^P \xrightarrow[\delta_P]{a} q_2^P$, then q_1^M is labelled with q_1^P. The abstraction of an automaton M wrt. the $\mathbb{B}_{\mathcal{P}}$ schema can be implemented analogously.

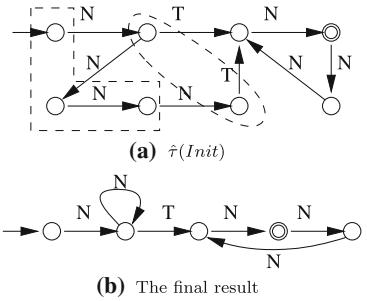
If the above construction is used, it is then clear that when *refining* $\mathbb{F}_{\mathcal{P}}$/$\mathbb{B}_{\mathcal{P}}$, we can just add X_k into $\mathcal{P}$ and modify the construction such that in the collapsing phase, we simply take into account all the labels by states of X_k and do not ignore the (anyway constructed) labels other than $q_0^{X_k}$.

Moreover, we can try to optimise the refinement of $\mathbb{F}_{\mathcal{P}}$/$\mathbb{B}_{\mathcal{P}}$ by replacing X_k in $\mathcal{P}$ by its *important tail/head part* defined wrt. M_k as the subautomaton of X_k based on the states of X_k that appear in at least one of the labels of M_k wrt. $\mathbb{F}_{\mathcal{P}\cup\{X_k\}}$/$\mathbb{B}_{\mathcal{P}\cup\{X_k\}}$, respectively. The effect of such a refinement corresponds to the weaker way of refining automata abstraction functions described in Sect. 4.2.2.[9] This is due to the strong link of the important tail/head part of X_k to M_k wrt. which it is computed. A repetition of the same faulty computation is then excluded, but the obtained abstraction is coarser, which may sometimes speed up the computation as we have already discussed.

A further possible heuristic to optimise the refinement of $\mathbb{F}_{\mathcal{P}}$/$\mathbb{B}_{\mathcal{P}}$ is trying to find just one or two *key states* of the important tail/head part of X_k such that if their languages are considered in addition to $\mathcal{P}$, $L(M_k^{\alpha})$ will not intersect $L(X_k)$.

We close the section by noting that in the *initial set of predicates* $\mathcal{P}$ of $\mathbb{F}_{\mathcal{P}}$/$\mathbb{B}_{\mathcal{P}}$, we may use, e.g., the automata describing the set of bad configurations and/or the set of initial configurations. Further, we may also use the domains or ranges of the transducers encoding the particular transitions in the systems being examined (whose union forms the one-step transition relation τ which we iterate). The meaning of the latter predicates is similar to using guards or actions of transitions in predicate abstraction [8].

[9] The key last/first jump in an accepting run of M mentioned in the proofs of Theorems 1, 2 is between states that can be labelled by some states of X. The concerned states of X are thus in the important tail/head part of X, and the proof construction of Theorems 1, 2 can still be applied.



(a) $\hat{\tau}(Init)$

(b) The final result

Fig. 7 An example using abstraction based on languages of words up to length n (for $n = 2$)

4.4 State equivalences using finite-length languages

We now present the approach of defining automata state equivalence schemas which is based on comparing automata states wrt. a certain bounded part of their languages. It is a simple, yet (according to our practical experience) often quite efficient approach. As a basic representative of this kind of schemas, we first present the schema $\mathbb{F}_n^L$ based on forward languages of words of a limited length. Then, we discuss its possible alternatives.

The $\mathbb{F}_n^L$ automata state equivalence schema defines two states of an automaton to be equal if their *languages of words of length up to a certain bound n* are identical. Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, $\mathbb{F}_n^L$ defines the state equivalence as the equivalence $\sim_M^n$ such that $\forall q_1, q_2 \in Q$: $q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

$\mathbb{F}_n^L$ is clearly *finitary*. It may be *refined* by incrementally increasing the bound n on the length of the words considered. This way, since we work with minimal deterministic automata, we may achieve the weaker type of refinement described in Sect. 4.2.2. Such an effect is achieved when n is increased to be equal or bigger than the number of states in M_k from Fig. 5 minus one. In a minimal deterministic automaton, this guarantees that all states are distinguishable wrt. $\sim_M^n$, and M_k will not be collapsed at all.

In Fig. 7, we apply $\mathbb{F}_n^L$ to the example from Fig. 4. We choose $n = 2$. In this case, the abstraction of the $Init$ automaton is $Init$ itself. Fig. 7a indicates the states of $\hat{\tau}(Init)$ that have the same languages of words up to size 2 and are therefore equivalent. Collapsing them yields the automaton shown in Fig. 7b (after determinisation and minimisation), which is a fixpoint. Notice that it is a different overapproximation of the set of reachable configurations than the one obtained using $\mathbb{F}_{\mathcal{P}}$. If we choose $n = 1$, we obtain a similar result, but we need one refinement step of the above described kind.

Let us, however, note that according to our practical experience, the increment of n by $|Q_M| - 1$ may often be too big. Alternatively, one may use a fraction of it (e.g., one half),

increase n by the number of states in X_k (or a fraction of it), or increase n just by one. In such cases, an immediate exclusion of the faulty run is not guaranteed, but clearly, such a computation will be *eventually excluded* because n will sooner or later reach the necessary value. The impact of working with abstractions refined in a coarser way is then like in the case of using $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$.

Regarding the *initial value of n*, one may use, e.g., the number of states in the automaton describing the set of initial configurations or the set of bad configurations, their fraction, or again just one.

As a natural alternative to dealing with forward languages of words of a limited length, one may also use *backward languages of words* of a limited length and forward/backward *languages of traces* of a limited length. The automata equivalence schemas $\mathbb{B}^L_n$, $\mathbb{F}^T_n$, as well as $\mathbb{B}^T_n$ based on them can be formally defined analogously to $\mathbb{F}^L_n$.

Clearly, all these schemas are *finitary*. Moreover, we can *refine* them in a similar way as $\mathbb{F}^L_n$. For $\mathbb{F}^T_n$ and $\mathbb{B}^T_n$, however, no guarantee of excluding a spurious counterexample may be provided. Using $\mathbb{F}^T_n$, e.g., we can never distinguish the last three states of the automaton in Fig. 7b—they all have the same trace languages. Thus, we cannot remember that the token cannot get to the last process. Nevertheless, despite this, our practical experience shows that the schemas based on traces may be quite successful in practice.

4.5 Experiments with abstract regular model checking

We have implemented the ideas described above in a prototype tool written in YAP Prolog using the FSA library [53]. To demonstrate that ARMC is applicable to verification of a broad variety of systems, we tried to apply the tool to a number of different verification tasks.

4.5.1 The types of systems verified

Parameterised networks of processes We have considered several somewhat idealised *mutual exclusion algorithms* for arbitrarily many processes (namely, the Bakery, Burns, Dijkstra, and Szymanski algorithms in versions similar to [41]). In most of these systems, the particular processes are finite-state. We encode their global configurations by words whose length corresponds to the number of participating processes, and each letter represents the local state of some process. In the case of the Bakery algorithm where each process contains an unbounded ticket value, this value is not represented directly, but encoded in the ordering of the processes in the word.

We verified the mutual exclusion property of the algorithms, and for the Bakery algorithm, we verified that some process will always eventually get to the critical section (communal liveness) as well as that each individual process will always eventually get there (individual liveness) under suitable fairness assumptions. For checking liveness, we manually composed the appropriate Büchi automata with the system being verified. Loop detection was allowed by working with pairs of configurations consisting of a remembered potential beginning of a loop (fixed at a certain—randomly chosen—point of time) and the current configuration. Checking that a loop is closed then consisted in checking that a pair of the same configurations was reached. To encode the pairs of configurations using finite automata, we interleaved their corresponding letters.

Push-down systems We considered a simple system of *recursive procedures*—the plotter example from [28]. We verified a safety part of the original property of interest describing the correct order of plotter instructions to be issued. In this case, we use words to encode the contents of the stack.

Systems with queues We experimented with a model of the Alternating Bit Protocol (ABP) for which we checked correctness of the delivery order of the messages. A word encoding a configuration of the protocol contained two letters representing internal states of the communicating processes. Moreover, it contained the contents of the two *lossy communication channels* with a letter corresponding to each message. Let us note that in this case, as well as in the above and below cases, general (non-length-preserving) transducers were used to encode transitions of the systems.

Petri nets and systems with counters We examined a general *Petri net* with inhibitor arcs, which can be considered an example of a system with *unbounded counters* too. In particular, we modelled a Readers/Writers system extended with a possibility of dynamic creation and deletion of processes, for which we verified mutual exclusion between readers and writers and between multiple writers. We considered a correct version of the system as well as a faulty one, in which we omitted one of the Petri net arcs. Markings of places in the Petri net were encoded in unary, and the particular values were put in parallel.[10] Further, we also considered the Bakery algorithm for two processes modelled as a counter automaton with two unbounded counters. For the actual verification, a binary encoding of the values of counters like in NDDs [59] was successfully used.

Dynamic linked data structures As a representative case study, we considered verification of a *procedure for reversing* (*non-empty*) *singly linked lists*—cf. Fig. 8.

[10] Using this encoding, a marking of a net with places p and q, two tokens in p, and four in q is written as $q|q|pq|pq$.

```
1: x = NULL;
2: while (list → next) {
3:     y = list → next;
4:     list → next = x;
5:     x = list;  list = y;
6: }
7: list → next = x;
```

Fig. 8 Nonempty list reversal

When abstracting the memory manipulated by the procedure, we focused on the cases where in the memory there are at most two linked lists linking consecutive cells, the first list in a descending way and the second one in an ascending way. We represented configurations of the procedure as words over the following alphabet: List items were represented by symbols $\underline{i}$, left/right pointers by </>, pointer variables were represented by their names ($list$ is shortened to l), and $\underline{o}$ was used to represent the memory outside the list. Moreover, we used symbols $\underline{iv}$ (resp. $\underline{ov}$) to denote that v points to i (resp. outside the list). We used | to separate the ascending and descending lists. Pointer variables pointing to null were not present in the configuration representations. A typical abstraction of the memory then looked like $\underline{i} < \underline{i} < \underline{i} \mid \underline{il} > \underline{i}\ \underline{ox}$ where the first list contains three items, the second one two, $list$ points to the beginning of the second list, x points outside the two lists, and y points to null. For such an abstraction of the memory contents (prefixed with the current control line), it is not difficult to associate transducers with each command of the procedure. For example, the transducer corresponding to the command $list \rightarrow next := x$ at line 4 transforms a typical configuration 4 $\underline{i} < \underline{ix} \mid \underline{il} > \underline{iy} > \underline{i}\ \underline{o}$ to the configuration 5 $\underline{i} < \underline{ix} < \underline{il} \mid \underline{iy} > \underline{i}\ \underline{o}$ (the successor of the item pointed to by l is not anymore the one pointed to by y, but the one pointed to by x). Then, the transducer τ corresponding to the whole procedure is the union of the transducers of all the commands.

If the memory contents did not fit the above described form, it would be abstracted to a single word with the "don't know" meaning. However, starting from configurations like 1 $\underline{il} > \underline{i} > \underline{i}\ \underline{o}$ or 1 $\underline{i} < \underline{i} < \underline{il}\ \underline{o}$, the verification showed that such a situation could not happen. Via a symmetry argument exploiting the fact that the procedure never refers to concrete addresses, the results of the verification easily generalise to lists with items stored at arbitrary memory locations.

By computing an abstraction of the reachability set $\tau^*(Init)$, we checked that the procedure outputs a list. Moreover, by computing an overapproximation of the reachability relation τ^* of the system, we checked that the output list has the same length as the input one.

In [12], a generalised encoding for 1-selector linked structures was provided and various list-manipulating procedures were successfully verified. Moreover, later, abstract regular *tree* model checking was used in [14] for verification of programs with dynamic data structures with more selectors and various complicated topologies. In this paper in Sect. 6, we concentrate on the latter, more recent and more general approach.

Table 1 Results of experimenting with abstract regular model checking using the finite-length-languages-based abstractions

Experiment	$\mathbb{F}_n^L/\mathbb{F}_n^T/\mathbb{B}_n^L/\mathbb{B}_n^T$	T_{best}
Bakery	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert/2$	0.02
Bakery/comm. liv.	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert$	0.14
Bakery/ind. liv.	Fw, $\mathbb{F}_n^T$, 1	8.66
Bakery—counters	Bw, $\mathbb{B}_n^L$, $\lvert Q_{Bad}\rvert$	0.08
ABP	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.32
Burns	Fw, $\mathbb{B}_n^T$, 1	0.31
Dijkstra	Fw, $\mathbb{F}_n^T$, 1	1.75
PDS	Bw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.02
Petri net/Read. Wr.	Fw, $\mathbb{B}_n^T$, special n	21.07
Faulty PN/Rd. Wr.	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert$	0.73
Szymanski	Fw, $\mathbb{B}_n^T$, 1	0.25
Rev. Lists	Fw,$\mathbb{F}_n^L$, $\lvert Q_{Init}\rvert/2 + \lvert Q_{X_k}\rvert/2$	0.61
Rev. Lists/Transd.	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Init}\rvert/2$	21.79

4.5.2 A summary of the results

The efficiency of using the $\mathbb{F}_n^L$, $\mathbb{F}_n^T$, $\mathbb{B}_n^L$, or $\mathbb{B}_n^T$ automata state equivalence schemas heavily depends on the *choice of the initial value of n* and the *strategy of increasing it*. In our experiments, we have tried $|Q_{Bad}|$, $|Q_{Bad}|/2$, $|Q_{Init}|$, $|Q_{Init}|/2$, and 1 as the initial value of n and $|Q_{M_k}|$, $|Q_{M_k}|/2$, $|Q_{X_k}|$, $|Q_{X_k}|/2$, and 1 as its increment. The results we obtained are summarised in Table 1. In the table, we always mention the scenario for which we obtained the shortest execution time.[11] We first say whether it was in a forward or backward computation (i.e., starting from the initial configurations or the "bad" ones); then the automata state equivalence schema used, followed by the initial value of n, and if it was needed, the increment of n written behind a plus symbol. In the case of the Readers/Writers example, the time consumption was relatively high, and we tried to iteratively find a value of n for which it was the best.

Similarly to the above, the efficiency of using the $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ automata state equivalence schemas depends a lot on the choice of the *initial predicates*. As the basic initial predicates in our experiments, we considered using automata representing the set of bad or initial configurations. We used them alone or together with automata corresponding to the domains or ranges of the transducers encoding the particu-

[11] In some cases, a few scenarios gave a very similar result out of which just one is mentioned.

Table 2 Results of experimenting with abstract regular model checking using the predicate-based abstractions

Experiment	$\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$	T_{best}
Bakery	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.02
Bakery/comm. liv.	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad\|Grd]$	0.13
Bakery/ind. liv.	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$, Key St.	19.41
Bakery—counters	Bw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\|Grd]$	0.09
ABP	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\|Grd]$	0.68
Burns	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.06
Dijkstra	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.73
PDS	Bw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.02
Petri net/Read. Wr.	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\|Grd]$	5.86
Faulty PN/Rd. Wr.	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\|Grd]$	0.81
Szymanski	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Init\|Grd]$	0.55
Rev. Lists	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\|Grd\|Act]$	1.29
Rev. Lists/Transd.	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\|Grd\|Act]$	42.60

lar transitions in the systems being examined. The scenarios that lead to the best results are listed in Table 2. The *heuristic optimisation* of the refinements described in Sect. 4.3.3 had a very significant positive impact in the case of checking individual liveness in the Bakery example. In the other cases, the effect was neutral or negative.

The times presented in Tables 1 and 2 are in seconds and were obtained on a computer with a 1.7 GHz Intel Pentium 4 processor. They do not include the time needed for reading the input model. Taking into account that the tool used was an early prototype written in YAP Prolog using the FSA library [53],[12] the results are very positive. For example, the Uppsala Regular Model Checker [5] took from about 8 to 11 seconds when applied to a comparable encoding of the Burns, Szymanski, and Dijkstra examples (and the situation did not change much with [42]). Finally, Tables 1 and 2 also show that apart from cases where the approaches based on languages of words/traces up to a bounded length and the ones based on intersections with predicate languages are roughly equal, there are really cases where either the former or the latter approach is faster. This experimentally justifies our interest in both of the techniques.

5 Regular tree model checking

As was already noted, regular tree model checking is a generalisation of regular (word) model checking to trees. A configuration of a system is encoded as a term (tree) over a ranked alphabet and a set of such terms as a regular tree automaton. The transition relation of a system is typically encoded as a linear tree transducer τ.[13]

To illustrate the use of tree automata and transducers, let us consider a simple example—namely, a generalisation of the simple token passing protocol from Sect. 3.1 to trees. We suppose having a tree-shaped network of processes of an arbitrary size. Initially, a token is situated in one of the leaf nodes. Then, it is to be sent up to the root. We would like to check that the token does not disappear nor duplicate.

The initial configurations of the simple tree token passing protocol is encoded by the tree automaton $Init = (Q_{Init}, \Sigma, F_{Init}, \delta_{Init})$ where $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{T_0, N_0\}$ and $\Sigma_2 = \{T, N\}$,[14] $Q_{Init} = \{p_0, p_1\}$, $F_{Init} = \{p_1\}$, and δ_{Init} contains the following transitions:

$$N_0 \to p_0 \qquad T_0 \to p_1$$
$$N(p_0, p_0) \to p_0$$
$$N(p_1, p_0) \to p_1 \quad N(p_0, p_1) \to p_1$$

The one-step transition relation is represented by the tree transducer τ with Σ used as the input/output alphabet, $Q_\tau = \{q_0, q_1, q_2\}$, $F_\tau = \{q_2\}$, and the following transitions[15]:

$$N_0/N_0 \to q_0 \qquad T_0/N_0 \to q_1$$
$$N/N(q_0, q_0) \to q_0 \quad T/N(q_0, q_0) \to q_1$$
$$N/T(q_1, q_0) \to q_2 \quad N/T(q_0, q_1) \to q_2$$
$$N/N(q_2, q_0) \to q_2 \quad N/N(q_0, q_2) \to q_2$$

Finally, the set of bad configurations is encoded by the tree automaton Bad with Σ as its ranked alphabet, $Q_{Bad} = \{r_0, r_1, r_2\}$, $F_{Bad} = \{r_0, r_2\}$, and the following transitions:

$$N_0 \to r_0 \qquad T_0 \to r_1$$
$$N(r_0, r_0) \to r_0 \qquad T(r_0, r_0) \to r_1$$
$$N(r_1, r_0) \to r_1 \qquad N(r_0, r_1) \to r_1$$
$$T(r_1, r_0) \to r_2 \qquad T(r_0, r_1) \to r_2$$
$$N \text{ or } T(r_1, r_1) \to r_2 \qquad N \text{ or } T(r_0 \text{ or } r_1, r_2) \to r_2$$
$$N \text{ or } T(r_2, r_0 \text{ or } r_1) \to r_2 \quad N \text{ or } T(r_2, r_2) \to r_2$$

Similarly to the case of classical word regular model checking, the basic *safety verification problem of regular tree model checking* consists in deciding whether $\varrho_\tau^*(L(Init)) \cap L(Bad) = \emptyset$ holds. Of course, this problem is again in general undecidable, an iterative computation of $\varrho_\tau^*(L(Init))$ does not necessarily terminate, and so some acceleration techniques are needed to make it terminate as often as possible. Generalisations of the various acceleration schemes

[12] Prolog was chosen as a rapid, but still relatively efficient, prototyping environment.

[13] Like in RMC, another possibility is to use several transducers and/or special-purpose operations on tree automata.

[14] To respect the formal definition of a ranked alphabet, we distinguish leaf and non-leaf nodes with/without a token.

[15] We are dealing with a relabelling transducer and for a better readability, we write its transitions in the form $f/g(q_1, q_2) \to q$ where f is an input symbol and g an output symbol.

from regular model checking into trees have been considered—see, e.g., [6,7,18,19,50,51]. Below, we concentrate on a generalisation of using abstraction for this purpose.

5.1 Abstract regular tree model checking

A generalisation of ARMC to trees was originally considered in [13]. The proposed approach allows one to deal with structure-preserving as well as non-preserving tree transducers. Similarly to the word case, the introduction of an *automated abstraction* with a counterexample-guided refinement brings in not only an efficient acceleration technique, but also a quite efficient way for fighting the state explosion problem in the number of tree automata states.

In particular, two abstractions for tree automata have been proposed. Similarly to abstract word regular model checking, both of them are based on collapsing automata states according to a suitable equivalence relation. The first is based on considering two tree automata states equivalent if their *languages of trees up to a certain fixed height* are equal. The second abstraction is defined by a set of *regular tree predicate languages* as an analogy to the word automata predicate abstraction.

The proposed technique was successfully applied for verification of parametric tree networks of processes [13] and also programs with complex dynamic data structures [14], which we will discuss in detail in Sect. 6.

5.1.1 The framework of ARTMC

We can formalise the basic framework of abstract regular tree model checking (ARTMC) in a way quite similar to word regular model checking. We basically phrase all the needed concepts not for classical finite automata, but for finite tree automata.

Note that in the following as in ARMC, in order to shorten the descriptions, we *identify a tree transducer and the relation it represents* and write $\tau(L)$ instead of $\varrho_\tau(L)$. Let $\iota \subseteq T_\Sigma \times T_\Sigma$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations $\tau^0 = \iota$, $\tau^{i+1} = \tau \circ \tau^i$ and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Below, we suppose $\iota \subseteq \tau$ meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

Let Σ be a ranked alphabet and $\mathbb{M}_\Sigma$ the set of all tree automata over Σ. We define an abstraction function as a mapping $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. Given a tree transducer τ and an abstraction α, we define a mapping $\tau_\alpha : \mathbb{M}_\Sigma \to \mathbb{M}_\Sigma$ as $\forall M \in \mathbb{M}_\Sigma : \tau_\alpha(M) = \hat{\tau}(\alpha(M))$ where $\hat{\tau}(M)$ is a minimal automaton describing the language $\tau(L(M))$. An abstraction α is *finite range* if the set $\mathbb{A}_\Sigma$ is finite.

Let $Init$ be a tree automaton representing the set of initial configurations and Bad be a tree automaton representing the set of bad configurations. Now, we may iteratively compute the sequence $(\tau_\alpha^i(Init))_{i\geq 0}$. Since we suppose $\iota \subseteq \tau$, it is clear that if α is finitary, there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(Init) = \tau_\alpha^k(Init)$. The definition of α implies $L(\tau_\alpha^k(Init)) \supseteq \tau^*(L(Init))$. This means that in a finite number of steps, we can compute an overapproximation of the reachability set $\tau^*(L(Init))$.

If $L(\tau_\alpha^k(Init)) \cap L(Bad) = \emptyset$, then the safety verification problem checking whether $\tau^*(L(Init)) \cap L(Bad) = \emptyset$ has a positive answer. Otherwise, the answer is not necessarily negative since during the computation of the set $\tau_\alpha^*(L(Init))$, the abstraction α may introduce extra behaviours leading to $L(Bad)$. Let us examine this case. Assume $\tau_\alpha^*(Init) \cap L(Bad) \neq \emptyset$, meaning that there is a symbolic path $Init,\ \tau_\alpha(Init),\ \tau_\alpha^2(Init),\ \ldots,\ \tau_\alpha^n(Init)$ such that $L(\tau_\alpha^n(Init)) \cap L(Bad) \neq \emptyset$. We analyse this path by computing the sets $X_n = L(\tau_\alpha^n(Init)) \cap L(Bad)$, and for every $k \geq 0$, $X_k = L(\tau_\alpha^k(Init)) \cap \tau^{-1}(X_{k+1})$. Two cases may occur: (1) $X_0 = L(Init) \cap (\tau^{-1})^n(X_n) \neq \emptyset$, which means that the safety verification problem has a *negative answer*, or (2) there is a $k \geq 0$ such that $X_k = \emptyset$, and this means that the considered symbolic path is actually a *spurious counterexample* due to the fact that α is too coarse. In this last situation, we need to refine α and iterate the procedure. Therefore, ARTMC is based on abstraction schemas allowing to compute families of (automatically) refinable abstractions.

5.1.2 Abstractions over tree automata

Below, we discuss two tree automata abstraction schemas based on tree automata state equivalences. First, tree automata states are split into several equivalence classes by an equivalence relation. Then, states from each equivalence class are collapsed into one state. Formally, a tree automata state equivalence schema $\mathbb{E}$ is defined as follows: To each tree automaton $M = (Q, \Sigma, F, \delta) \in \mathbb{M}_\Sigma$, an equivalence relation $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ is assigned. Then the automata abstraction function $\alpha_{\mathbb{E}}$ corresponding to the abstraction schema $\mathbb{E}$ is defined as $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. We call $\mathbb{E}$ finitary if $\alpha_{\mathbb{E}}$ is finitary (i.e., there is a finite number of equivalence classes). We refine $\mathbb{E}$ by making $\sim_M^{\mathbb{E}}$ finer.

Abstraction based on tree languages of finite height We now present the possibility of defining automata state equivalence schemas which are based on comparing automata states wrt. a certain bounded part of their languages. The abstraction schema $\mathbb{H}_n$ is a generalisation of the schema based on languages of words up to a certain length (cf. Sect. 4.4). The $\mathbb{H}_n$ schema defines two states of a tree automaton M as equivalent if their languages up to the given height n are identical.

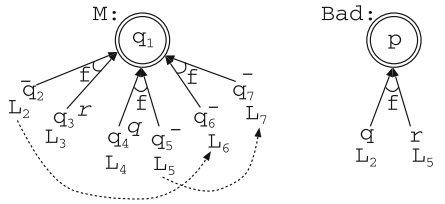


Fig. 9 A problem with the forward tree predicate abstraction

Formally, for a tree automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{H}_n$ defines the state equivalence as the equivalence $\sim_M^n$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

There is a finite number of languages of trees with a maximal height n, and so this abstraction is finite range. Refining of the abstraction can be done by increasing the value of n.

One can implement the abstraction schema $\mathbb{H}_n$ much like minimisation of tree automata [22], by simply stopping the main minimisation loop after n iterations.

Abstraction based on predicate tree languages We next introduce a predicate-based abstraction schema $\mathbb{P}_\mathcal{P}$ that is inspired by the predicate-based abstraction on words discussed in Sect. 4.3.

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $M = (Q, \Sigma, F, \delta)$ be a tree automaton, then two states $q_1, q_2 \in Q$ are equivalent if their languages $L(M, q_1)$ and $L(M, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$. Formally, for an automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{P}_\mathcal{P}$ defines the state equivalence as the equivalence $\sim_M^\mathcal{P}$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^\mathcal{P} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(M, q_2) \neq \emptyset)$.

Clearly, since $\mathcal{P}$ is finite and there is only a finite number of subsets of $\mathcal{P}$ representing the predicates with which a given state has a nonempty intersection, $\mathbb{P}_\mathcal{P}$ is *finitary*. It can be refined by adding new predicates into $\mathcal{P}$ in a way analogous to the word case (cf. Sect. 4.3). Thus, we can show that a spurious counterexample can be eliminated by extending the predicate set $\mathcal{P}$ by the languages of all states of the tree automaton representing X_{k+1} in the analysis of the spurious counterexample (recall that $X_k = \emptyset$) as presented in Sect. 5.1. Similar optimisations like those in Sect. 4.3.3 apply here too.

Above, we discussed the $\mathbb{P}_\mathcal{P}$ abstraction schema inspired by the predicate-based abstraction from word ARMC. In particular, it is inspired by the *backward* predicate-based abstraction schema $\mathbb{B}_\mathcal{P}$. Interestingly, as illustrated in Fig. 9, it is impossible to obtain a tree analogy with the *forward* predicate-based abstraction schema $\mathbb{F}_\mathcal{P}$ of word ARMC. The tree analogy would be to label a state with a predicate state if the languages of their contexts—i.e., trees where we substitute Σ^* for the language of the node being labelled/used for labelling—have a non-empty intersection. However, in this case, the refinement schema we use in all our predicate-based abstractions does not work. For instance, consider tree automata whose fractions are shown in Fig. 9. In the figure, L_i below a state means that the language of that state is L_i, and we assume that $L_i \cap L_j = \emptyset$ for any $i \neq j$. Suppose we start with no predicates and want to refine the abstraction so that the refined abstraction of M does not intersect the language of Bad. To ensure this using our refinement schema, we should take the context languages of the states of Bad as the new predicates. Assume we do so and try to abstract M. In Fig. 9, the upper index of the states of M shows by which states of Bad they are labelled when abstracting M (the minus sign means that no state appears in the label). The abstraction would now collapse states q_2, q_5, q_6, and q_7. Consequently, as the arrows show, the resulting automaton could accept trees $f(t_1, t_2)$ for $t_1 \in L_2$ and $t_2 \in L_5$ belonging to $L(Bad)$ despite $L(M)$ does not contain such trees. So, the refinement has not excluded trees from $L(Bad)$ from the language of the abstraction of M and there is no way how to refine the abstraction further using our refinement schema.

6 Verification of programs with pointers

This section discusses our fully automated method for analysing various important properties of programs manipulating *complex dynamic linked data structures* that was first published in [14]. We consider non-recursive, sequential C programs manipulating dynamic linked data structures with possibly several next pointer selectors, storing data from finite domains. The properties to be checked include basic consistency of pointer manipulations (i.e., checking that there are no *null pointer assignments*, no use of *undefined pointers*, and no references to *deleted elements*). Moreover, further undesirable behaviours of the programs at hand (such as, for instance, disobedience of certain *shape invariants*, e.g., due to introducing undesirable sharing, cycles, etc.) may be detected via *testers written in C* and attached to the verified programs. Then, verification of such properties reduces to reachability of a designated error location.

Our verification method uses the approach of *abstract regular tree model checking*. In order to be able to apply it on programs manipulating dynamic linked data structures whose configurations (naturally viewed as the so-called *shape graphs*) need not be tree-like, we proceed as follows. We use trees to encode the *tree skeletons* of shape graphs. The edges of a shape graph that are not directly encoded in the tree skeleton are represented by *routing expressions* over the tree skeleton—i.e., regular expressions over directions in a tree (such as left up, right down, etc.) and the kind of nodes that can be visited on the way. The routing expressions are referred to from the tree skeletons. Both the tree skeletons and the routing expressions are automatically discovered by

our method. The idea of using routing expressions is inspired by PALE [40] and graph types [38].

We implemented our method in a prototype tool built on top of the Mona tree libraries [37]. We have tested it on a number of non-trivial procedures manipulating singly linked lists (SLL), doubly linked lists (DLL), trees (including the Deutsch–Schorr–Waite tree traversal), lists of lists, and also trees with linked leaves. All the procedures were automatically verified for absence of null pointer dereferences, absence of manipulation with undefined pointers, and absence of dereferencing of deleted objects. Additionally, further shape properties (such as absence of sharing, acyclicity, preservation of input elements, etc.) were also verified for some of the procedures.

6.1 Related approaches

The area of research on automated verification of programs manipulating dynamic linked data structures is very active. Various approaches to verification of such programs differing in their principles, degree of automation, generality, and scalability have emerged. They are based, e.g., on monadic second-order logic [40], 3-valued predicate logic with transitive closure [47], separation logic [20,31,45,60],[16] or automata [12,14,25]. Among all of these approaches, the method presented here is one of the most general and fully automated at the same time.

The closest approach to what we present here is the one of PALE that also uses tree automata (derived from WSkS formulae) as well as the idea of a tree skeleton and routing expressions. However, first, the encoding of PALE is different in that the routing expressions must deterministically choose their target, and also, for a given memory node, selector, and program line, the expression is fixed and cannot dynamically change during the run of the analysed program. Further, program statements are modelled as transformers on the level of WSkS formulae, not as transducers on the level of tree automata. Finally, the approach of PALE is not fully automatic as the user has to manually provide loop invariants and all needed routing expressions, which are automatically synthesised in our approach.

6.2 The considered programs

We consider standard, non-recursive, sequential C programs manipulating dynamic linked data structures with possibly several next pointer selectors. We do not consider pointer arithmetics, and we suppose all non-pointer data to be abstracted to a finite domain by some of the existing techniques before our method is applied. The abstract syntax of the considered programs is given in Fig. 10a, where Lab is a finite set of program labels (one for each control location), $\mathcal{V}$ is a finite set of pointer variables, $\mathcal{D}$ is a finite set of data values, and $\mathcal{S}$ is a finite set of selectors. We suppose other commonly used statements (such as `while` loops or nested dereferences) to be encoded by the listed statements. An example of a typical program that our method can handle is the reversion of doubly linked lists (DLLs) shown in Fig. 10b, which we also use as our running example.

Memory configurations Memory configurations of the considered programs with a finite set of pointer variables $\mathcal{V}$, a finite set of selectors $\mathcal{S} = \{1, \ldots, k\}$, and a finite domain $\mathcal{D}$ of data stored in dynamically allocated memory cells can be described as shape graphs of the following form. A *shape graph* is a tuple $SG = (N, S, V, D)$ where N is a finite set of memory nodes, $N \cap \{\bot, \top\} = \emptyset$ (we use $\bot$ to represent null, and $\top$ to represent an undefined pointer value), $N_{\bot,\top} = N \cup \{\bot, \top\}$, $S : N \times \mathcal{S} \to N_{\bot,\top}$ is a successor function, $V : \mathcal{V} \to N_{\bot,\top}$ is a mapping that defines where the pointer variables are currently pointing to, and $D : N \to \mathcal{D}$ defines what data are stored in the particular memory nodes.

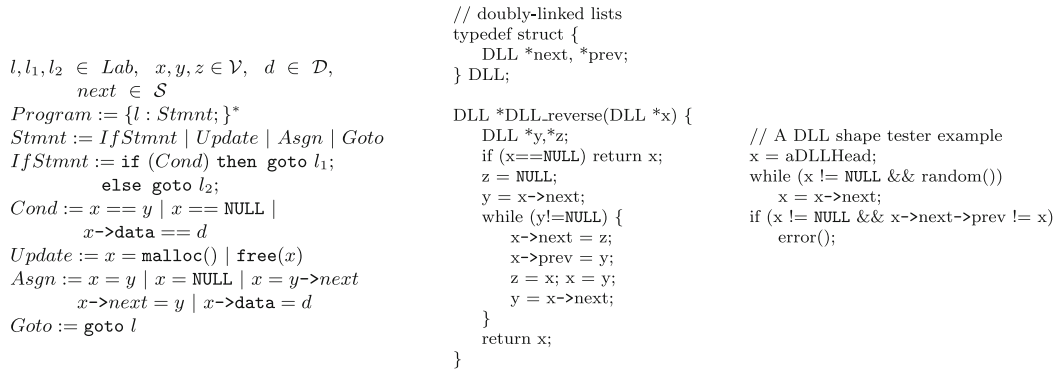
6.3 The considered properties

First of all, the properties we intend to check include *basic consistency of pointer manipulations*, i.e., absence of null and undefined pointer dereferences and references to already deleted nodes. Further, we would like to check various *shape invariance properties* such as absence of sharing, acyclicity, or, e.g., the fact that if `x->next == y` (and `y` is not null) in a DLL, then also `y->prev == x`, etc. To define such properties, we use the so-called *shape testers* written in the C language. They can be seen as instrumentation code trying to detect violations of the memory shape properties at selected control locations of the original program.

For defining testers, we slightly extend the C language by allowing next pointers to be followed backwards and by non-deterministic branching. The testers become a part of the code being verified. An error is announced when a line denoted by an error label is reached. This way, we can check a whole range of properties, including acyclicity, absence of sharing, and other shape invariants such as the relation of next and previous pointers in DLLs—cf. Fig. 10c. Shape testers can be directly written by the user, or they can be generated from a more declarative specification based, e.g., on the specialised logic proposed in [14].

In theory, bad shapes may be described directly using a tree automata memory encoding. The problem is to not miss any of their possible encodings since—as we will see—the memory encoding that we are going to use is not canonical. This problem does not arise when using shape testers as in their case, only reachability of a certain line is tested and

[16] We briefly comment on these approaches in Sect. 6.8 too.



(a)

$l, l_1, l_2 \in Lab, \quad x, y, z \in \mathcal{V}, \quad d \in \mathcal{D},$
$next \in \mathcal{S}$
$Program := \{l : Stmnt;\}^*$
$Stmnt := IfStmnt \mid Update \mid Asgn \mid Goto$
$IfStmnt :=$ `if` $(Cond)$ `then goto` $l_1;$
`else goto` $l_2;$
$Cond := x == y \mid x ==$ `NULL` $\mid$
x`->data` $== d$
$Update := x =$ `malloc()` $\mid$ `free`(x)
$Asgn := x = y \mid x =$ `NULL` $\mid x = y$`->`$next$
x`->`$next = y \mid x$`->data` $= d$
$Goto :=$ `goto` l

(b)

```
// doubly-linked lists
typedef struct {
    DLL *next, *prev;
} DLL;

DLL *DLL_reverse(DLL *x) {
    DLL *y,*z;
    if (x==NULL) return x;
    z = NULL;
    y = x->next;
    while (y!=NULL) {
        x->next = z;
        x->prev = y;
        z = x; x = y;
        y = x->next;
    }
    return x;
}
```

(c)

```
// A DLL shape tester example
x = aDLLHead;
while (x != NULL && random())
    x = x->next;
if (x != NULL && x->next->prev != x)
    error();
```

Fig. 10 (**a**) Abstract syntax of the considered programs. (**b**) A running example: reversion of DLLs. (**c**) A shape tester example

the choice of a suitable encoding is subject to the automatic abstraction refinement.

6.4 The verification problem

Above, we have explained that for checking preservation of shape invariants, we use shape testers, for which we need to check unreachability of their designated error location. Moreover, we model all program statements such that if some basic memory consistency error (like a null pointer assignment) happens, the control is automatically transferred to a unique error control location. Thus, we are in general interested in *checking unreachability of certain error control locations* in a program.

6.5 Encoding the programs in tree automata

In this section, we describe our encoding of memory configurations of the considered programs into trees and tree automata and our encoding of program statements using tree transducers and specialised automata operations.

6.5.1 Encoding of sets of memory configurations

As was described in Sect. 6.2, memory configurations of the considered programs with a finite set of pointer variables $\mathcal{V}$, a finite set of selectors $\mathcal{S} = \{1, \ldots, k\}$, and a finite domain $\mathcal{D}$ of data stored in dynamically allocated memory cells can be described as *shape graphs* $SG = (N, S, V, D)$. We suppose $\top \in \mathcal{D}$—the data value $\top$ is used to denote "zombies" of deleted nodes, which we keep and detect all erroneous attempts to access them.

To be able to describe the way we encode sets of shape graphs using tree automata, we first need a few auxiliary notions. First, to allow for dealing with more general shape graphs than tree-like, we do not simply identify the next pointers with the branches of the trees accepted by tree automata. Instead, we use the tree structure just as a backbone over which links between the allocated nodes are expressed using the so-called *routing expressions*, which are regular expressions over directions in a tree (like move up, move left down, etc.) and over the nodes that can be seen on the way. From nodes of the trees described by tree automata, we refer to the routing expressions via their symbolic names called *pointer descriptors*—we suppose dealing with a finite set of pointer descriptors $\mathcal{R}$. Moreover, we couple each pointer descriptor with a unique *marker* from a set $\mathcal{M}$ (and so $|\mathcal{R}| = |\mathcal{M}|$). The routing expressions may identify several target nodes for a single source memory node and a single selector. Markers associated with the target nodes can then be used to decrease the non-determinism of the description (only nodes marked with the right marker are considered as the target).

Let us now fix the sets $\mathcal{V}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{M}$. We use a *ranked alphabet* $\Sigma = \Sigma_2 \cup \Sigma_1 \cup \Sigma_0$ consisting of symbols of ranks $k = |\mathcal{S}|$, 1, and 0. Symbols of rank k represent allocated memory nodes or nodes that were allocated, but later they have been deleted (freed). Allocated nodes may be pointed to by pointer variables whereas deleted nodes are not pointed to by any variables since we make all variables pointing to such nodes undefined. Allocated as well as deleted nodes may be marked by some markers as targets of some next pointers, they contain some data, and have k next pointers that are either null, undefined (which is the only possibility for deleted nodes), or given by some next pointer descriptor. Thus, $\Sigma_2 = \Sigma_{2,a} \cup \Sigma_{2,d}$ where $\Sigma_{2,a} = 2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D} \times (\mathcal{R} \cup \{\bot, \top\})^k \times \{\texttt{alloc}\}$ and $\Sigma_{2,d} = \{\emptyset\} \times 2^{\mathcal{M}} \times \mathcal{D} \times \{\top\}^k \times \{\texttt{del}\}$. Given an element $n \in \Sigma_2$, we use the notation $n.var$, $n.mark$, $n.data$, and $n.s$ (for

$s \in \mathcal{S}$) to refer to the pointer variables, markers, data, and descriptors associated with n, respectively. Σ_1 is used for specifying nodes with undefined and null pointer variables, and so $\Sigma_1 = 2^{\mathcal{V}}$. Finally, in our trees, the leaves are all the same (with no special meaning), and so $\Sigma_0 = \{\bullet\}$.

We can now specify the *tree memory backbones* we use to encode memory configurations as the trees that belong to the language of the tree automaton with the following rules[17]: (1) $\bullet \rightarrow q_i$, (2) $\Sigma_2(q_i/q_m, \ldots, q_i/q_m) \rightarrow q_m$, (3) $\Sigma_1(q_m/q_i) \rightarrow q_n$, and (4) $\Sigma_1(q_n) \rightarrow q_u$. Intuitively, q_i, q_m, q_n, and q_u are automata states where q_i accepts the leaves, q_m accepts the memory nodes, q_n accepts the node encoding null variables, and q_u, which is the accepting state, accepts the node with undefined variables. Note that there is always a single node with undefined variables, a single node with null variables, and then a sub-tree with the memory allocated nodes. Thus, every memory tree t can be written as $t = \mathit{undef}(\mathit{null}(t'))$ for $\mathit{undef}, \mathit{null} \in \Sigma_1$. We say a memory tree $t = \mathit{undef}(\mathit{null}(t'))$ is *well-formed* if the pointer variables are assigned unique meanings, i.e., $\mathit{undef} \cap \mathit{null} = \emptyset \wedge \forall p \in \mathcal{N}lPos(t') : t'(p).var \cap (\mathit{null} \cup \mathit{undef}) = \emptyset \wedge \forall p_1 \neq p_2 \in \mathcal{N}lPos(t') : t'(p_1).var \cap t'(p_2).var = \emptyset$.

We let $\mathcal{S}^{-1} = \{s^{-1} \mid s \in \mathcal{S}\}$ be the set of "inverted selectors" allowing one to follow the links in a shape graph in the reverse order. A *routing expression* is then formally defined as a regular expression on pairs $s.p \in (\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma_2$. Intuitively, each pair used as a basic building block of a routing expression describes one step over the tree memory backbone: The step follows a certain branch up or down after which a certain kind of node should be encountered (most often, we will use the node components of routing expressions to check whether a certain marker is set in the target node).

A *tree memory encoding* is a tuple (t, μ) where t is a tree memory backbone and μ a mapping from the set of pointer descriptors $\mathcal{R}$ to routing expressions over the set of selectors $\mathcal{S}$ and the memory node alphabet Σ_2 of t. An example of a tree memory encoding for a *doubly linked list* (DLL) is shown in Fig. 11.

Let (t, μ), $t = \mathit{undef}(\mathit{null}(t'))$, be a tree memory encoding with a set of selectors $\mathcal{S}$ and a memory node alphabet Σ_2. We call $\pi = p_1 s_1 \ldots p_l s_l p_{l+1} \in \Sigma_2.((\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma_2)^l$ a *path in* t of length $l \geq 1$ iff $p_1 \in \mathcal{P}os(t')$ and $\forall i \in \{1, \ldots, l\} : (s_i \in \mathcal{S} \wedge p_i.s_i = p_{i+1} \wedge p_{i+1} \in \mathcal{P}os(t')) \vee (s_i \in \mathcal{S}^{-1} \wedge p_{i+1}.s_i = p_i)$. For $p, p' \in \mathcal{N}lPos(t')$ and a selector $s \in \mathcal{S}$, we write $p \xrightarrow{s} p'$ iff (1) $t'(p).s \in \mathcal{R}$, (2) there is a path $p_1 s_1 \ldots p_l s_l p_{l+1}$ in t for some $l \geq 0$ such that $p = p_1$, $p_{l+1} = p'$, and (3) $s_1 t'(p_2) \ldots t'(p_l) s_l t'(p_{l+1}) \in \mu(t'(p).s)$.

[17] We use a set of symbols instead of a single input symbol in a transition rule to concisely describe a set of rules using any of the symbols in the set. Similarly, a use of q_1/q_2 instead of a single state means that one can take either q_1 or q_2, and if there is a k-tuple of states, one considers all possible combinations of the states.

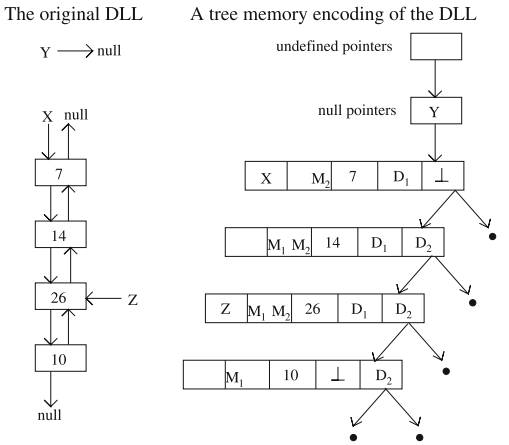


Fig. 11 A tree memory encoding for a doubly linked list (DLL). The descriptors are mapped as follows: $D_1 : 1.M_1$ and $D_2 : \overline{1}.M_2$. Only allocated nodes are present; hence the `alloc` flag is omitted

The *set of shape graphs represented by a tree memory encoding* (t, μ) with $t = \mathit{undef}(\mathit{null}(t'))$ is denoted by $[\![(t, \mu)]\!]$ and given as all the shape graphs $SG = (N, S, V, D)$ for which there is a bijection $\beta : \mathcal{P}os(t') \rightarrow N$ such that

1. $\forall p, p' \in \mathcal{N}lPos(t')\ \forall s \in \mathcal{S} : (t'(p).s \notin \{\bot, \top\} \wedge p \xrightarrow{s} p') \Leftrightarrow S(\beta(p), s) = \beta(p')$, i.e., the links between memory nodes are respected.
2. $\forall p \in \mathcal{N}lPos(t')\ \forall s \in \mathcal{S}\ \forall x \in \{\bot, \top\} : t'(p).s = x \Leftrightarrow S(\beta(p), s) = x$, i.e., null and undefined successors are respected.
3. $\forall v \in \mathcal{V}\ \forall p \in \mathcal{P}os(t') : v \in t'(p).var \Leftrightarrow V(v) = \beta(p)$, i.e., assignment of memory nodes to variables is respected.
4. $\forall v \in \mathcal{V} : (v \in \mathit{null} \Leftrightarrow V(v) = \bot) \wedge (v \in \mathit{undef} \Leftrightarrow V(v) = \top)$, i.e., assignment of null and undefinedness of variables are respected.
5. $\forall p \in \mathcal{N}lPos(t')\ \forall d \in \mathcal{D} : t'(p).data = d \Leftrightarrow D(\beta(p)) = d$, i.e., data stored in memory nodes is respected.

A *tree automata memory encoding* is a tuple (A, μ) where A is a tree automaton accepting a regular set of tree memory backbones and μ is a mapping as above. Naturally, A represents the set of shape graphs defined by $[\![(A, \mu)]\!] = \bigcup_{t \in L(A)} [\![(t, \mu)]\!]$.

The tree automata memory encoding is clearly *not canonical*, i.e. two tree automata having different languages might represent the same set of shape graphs. Nevertheless, as

we show below, program statements can still be encoded faithfully, partly using relabelling tree transducers and partly specialised operations on tree automata. Another important property of the encoding is that given a tree automata memory encoding (A, μ), the set $[\![(A, \mu)]\!]$ can be empty although $L(A)$ is not empty (since the routing expressions can be incompatible with the tree automaton). Of course, if $L(A)$ is empty, then $[\![(A, \mu)]\!]$ is also empty. Therefore, checking emptiness of $[\![(A, \mu)]\!]$ (which is important for applying the ARTMC framework, see Sect. 6.7) can be done in a sound way by checking emptiness of $L(A)$.

6.5.2 Pointer descriptors and routing expressions

As for the set of *pointer descriptors* $\mathcal{R}$, we restrict ourselves to a unique pointer descriptor for each destructive update `x->s = y` that appears in the program. This is because statements of this kind establish new links among the allocated memory nodes and having one descriptor per such a statement appears to be sufficient according to our practical experience. In addition, we might have some further descriptors if they are a part of the specification of the input configurations (see Sect. 6.7).

Further, in our automata-based framework, we encode *routing expressions* using tree transducers. A transducer representing a routing expression r simply copies the input tree memory backbone on which it is applied up to (1) looking for a data node n_1 that is labelled with a special token $\blacklozenge \notin \mathcal{V} \cup \mathcal{M} \cup \mathcal{D}$ and (2) moving $\blacklozenge$ to a data node n_2 that is the target of the next pointer described by r and that is also marked with the appropriate marker. As described in the next section, we can then implement program statements that follow the next pointers (e.g., `x = y->s`) by putting the token $\blacklozenge$ to a node pointed to by `y`, applying the transducer implementing the appropriate routing expression, and making `x` point to the node to which $\blacklozenge$ was moved. Due to applying abstraction, the target may not always be unique—in such a case, the transducer implementing the routing expression simply returns a set of trees in which $\blacklozenge$ is put to some target data node such that all possibilities where it can get via the given routing expression are covered.

Note that the use of tree transducers for encoding routing expressions allows us in theory to express more than using just regular expressions. In particular, we can refer to the tree context of the nodes via which the given route is going. In our current implementation, we, however, do not use this fact.

6.5.3 Encoding of program statements

We encode the considered pointer-manipulating statements as relabelling tree transducers or sets of such transducers being applied sequentially, in one case combined with an application of an additional specialised operation on the tree automata being handled.[18] When simulating the various program statements, we expect the tree memory encoding to be extended by a new root symbol, corresponding to the *current program line* or to an *error indicator* when an error is found during the analysis. The encoding of the program statements works in such a way that the effect of the statements is simulated on any set of shape graphs represented by a tree automata memory encoding. If a shape graph SG represented by a tree memory encoding is changed by a program statement to a shape graph SG', then the encoding of the statement transforms the tree memory encoding such that it represents SG'. This makes sure that although the memory encoding is not canonical, we simulate program statements faithfully.

Non-destructive updates and tests The simplest statement to encode is the `x = NULL` assignment. The transducer implementing it simply goes through the input tree and copies it to the output with the exception that (1) it removes `x` from the labelling of the node in which it currently is, (2) it adds `x` to the labelling of the *null* node, and (3) it changes the current line appropriately. The transducer implementing an assignment `x = y` is similar, it just puts `x` not to the *null* node, but to the node which is currently labelled by `y`.

The transducers encoding conditional statements of the form `if (x == NULL) goto l1; else goto l2;` are very similar to the above—of course, they do not change the node in which `x` is, but only change the current program line to either `l1` or `l2` according to whether `x` is in the *null* node or not. If `x` is in *undef*, an error indication is used instead of `l1` or `l2`. The transducers encoding statements `if (x == y) goto l1; else goto l2;` are similar—they test whether or not `x` and `y` appear in the same node (both being different from *undef*).

The transducer for an `x = y->s` statement is a union of several complementary actions. If `y` is in *null* or *undef*, an error is indicated. If `y` is in a regular data node and its s-th next pointer node contains either $\bot$ or $\top$, the transducer removes `x` from the node it is currently in and puts it into the *null* or *undef* node, respectively. If `y` is in a regular data node n and its s-th next pointer node contains some pointer descriptor $r \in \mathcal{R}$, the $\blacklozenge$ token is put to n. Then, the routing expression transducer associated with r is applied. Finally, `x` is removed from its current node and put into the node to which $\blacklozenge$ was moved by the applied routing expression transducer. If the target is marked as deleted, an error is announced.

Destructive updates Destructive pointer updates of the form `x->s = y` are implemented as follows: If `x` is in *null* or *undef*, an error is announced. If `x` is defined and `y` is in *null*

[18] The primary reason for this is to avoid a need of implementing non-structure preserving transducers on top of the MONA tree automata library [37], which we use to implement our techniques.

or *undef*, the transducer puts $\bot$ or $\top$ into the s-th next pointer node below x, respectively. Otherwise, the transducer puts the pointer descriptor r associated with the particular `x->`s `= y` statement being fired into the s-th next pointer node below x, and it marks the node in which y is by the marker coupled with r. Then, the routing expression transducer associated with r is updated such that it includes the path from the node of x to the node of y.

One could think of various strategies how to *extract the path* going from the node of x to the node of y. We consider a simple strategy, which is, however, successful in many practical examples as our experiments show. We extract the shortest path between x and y on the tree memory backbone, which consists of going some number of steps upwards to the closest common parent of x and y and then going some number of steps downwards. The upward or the downward phase can also be skipped when going just down or up, respectively. When extracting this shortest path, we project away all information about nodes we see on the way and about nodes not directly lying on the path. Only the directions (left/right up/down) and the number of steps are preserved.

Note that we, in fact, perform the operation of routing expression extraction on a tree automaton, and we extract all possible paths between where x and y may currently be. The result is transformed into a transducer τ_{xy} that moves the token ♦ from the position of x to the position of y, and τ_{xy} is then united with the current routing expression transducer associated with the given pointer descriptor r. The extraction of the routing paths is done partly by rewriting the input tree automaton via a special transducer τ_π that in one step identifies all the shortest paths between all x and y positions and projects away the non-necessary information about the nodes on the way. The transducer τ_π is simple. It just checks that one follows some branch up from x and then some branch down to y where the up and down sweeps meet in a single node. The transition relation of the resulting transducer is then post-processed by changing the context of the path to an arbitrary one, which is done by directly modifying the structure of the transducer.[19]

Dynamic allocation and deallocation Statements of the form `x = malloc()` are implemented by rewriting the right-most • leaf node to a new data node pointed to by x. All the k next pointers are set to $\top$.

To be able to exploit the regularity that is mostly present in algorithms allocating new data structures, which typically add new elements at the end/leaves of the structure, we also explicitly support a statement of the form `x.`s `= malloc()`. We even try to pre-process programs and compact all successive pairs of statements of the form `x = malloc(); y->`s `= x` (provided x is not used any further) to `y->`s`= malloc()`. This statement is then implemented by adding the new element directly under the node pointed to by y (provided it is a leaf) and joining it by a simple routing expression of the form "one level down via a certain branch". This typically yields much simpler and more precise routing expressions.

Finally, statements of the form `free(x)` are implemented by transducers that move all variables that are currently in the node pointed to by x to the *undef* node (if x is in *null* or *undef*, an error is announced). Then, the node is denoted as deleted, but it stays in our tree memory encoding with all its current markers set.

6.6 Input structures for the verified programs

In order to encode the input structures, we can directly use the tree automata memory encoding. Such an encoding can be provided manually or derived automatically from a description of the concerned linked data structure provided, e.g., as a graph type [38]. The main advantage is that the verification process starts with an exact encoding of the set of all possible instances of the considered data structure. Another possibility is to attach a *constructor written in C* before the verified procedure. The verification then starts with the empty shape graph.

6.7 Applying ARTMC

Apparently, we assume ARTMC to be used in its more general form, having the one-step transition relation split into several transducers that are applied in some particular order, together with one special operation on the tree automata used when extracting the routing expressions. We compute an overapproximation of the reachable configurations for each program line in such a way that we start from an initial set of shape graphs represented by a tree automata memory encoding (possibly representing the empty heap when an input constructor is used) and we iterate the abstract fixpoint computation described in Sect. 5.1 along the control flow graph of the program (using the depth-first strategy). The fixpoint computation stops if the abstraction α that is used is finitary. In such a case, the number of abstracted tree automata that encode sets of memory backbones which can arise in the program being checked is finite. Moreover, the number of the arising routing expressions is also finite since they are extracted from the bounded number of tree automata describing the encountered sets of memory backbones.[20]

[19] A more precise, but also more costly, approach would be to preserve (some of) the context.

[20] The non-canonicity of our encoding does not prevent the computation from stopping. It may just take longer since several encodings for the same graph could be added.

During the computation, we check whether a designated error location in the program is reached, a basic pointer exception is detected during simulating the effect of some statement, or whether a fixpoint is attained. In the latter case, the program is found correct. In the former case, we compute backwards along the path in the CFG that is being currently explored to check if the obtained counterexample is spurious as explained in Sect. 5.1. However, as said in Sect. 6.5.1, the check for emptiness is not exact and therefore we might conclude that we have obtained a real counterexample although this is not the case. However, such a situation has never happened in any of our experiments.[21]

We use a slight refinement of the basic finite-height and predicate abstractions described in Sect. 5.1. Concretely, we prevent the abstraction from allowing a certain pointer variable to point to several memory nodes at the same time. In particular, this amounts to prohibiting collapsing of states that would create a loop over a node pointed to by some pointer variable.

Apart from the basic abstraction schemas, we support one more abstraction schema called the *neighbour abstraction*. Under this schema, only the tree automata states are collapsed that (1) accept nodes with equal labels and (2) that directly follow each other (i.e., they are neighbours). This strategy is very simple, yet it proved useful in some practical cases.

Finally, we allow the abstraction to be applied either at all program lines or only at the loop closing points. In some cases, the latter approach is more advantageous due to that some critical destructive pointer updates are done without being interleaved with abstraction. This way, we may avoid having to remove lots of spurious counterexamples that may otherwise arise when the abstraction is applied while some important shape invariant is temporarily broken.

6.8 Experimental results

We have implemented the above proposed method in a prototype tool[22] based on the Mona tree automata library [37]. We have performed a set of experiments with singly linked lists (SLL), doubly linked lists (DLL), trees, lists of lists, and trees with linked leaves. As one of the most complicated case studies, we have also considered the so-called *task-lists*. The task-list structure is showed in Fig. 12 and it is inspired by the structures often used in operating systems [9].

All three mentioned types of automata abstraction—the finite-height abstraction (with the initial height being one), predicate abstraction (with no initial predicates), and neighbour abstraction—proved useful in different experiments. All case studies were automatically verified for

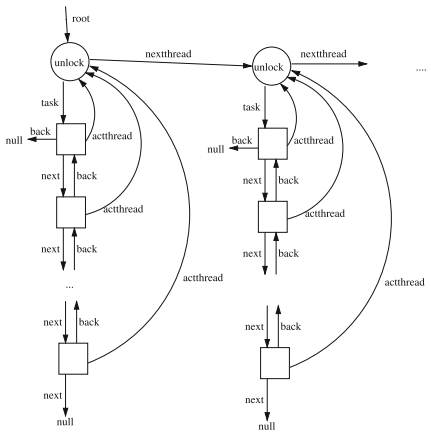


Fig. 12 The "task-list" data structure

null/undefined/deleted pointer exceptions. Additionally, some further shape properties (such as absence of sharing, acyclicity, preservation of input elements, etc.) were verified in some case studies too. For a detailed overview of the case studies and verified properties, see [46].

Table 3 contains verification times for our experiments. We give the best result obtained using one of the three mentioned abstraction schemas and say for which schema the result was obtained. The note "restricted" accompanying the abstraction means that the abstraction was applied at the loop points only. The experiments were performed on a 64 bit Opteron at 2.8 GHz. The column $|Q|$ gives information about the size (in numbers of states) of the biggest encountered automaton while N_{ref} gives the number of refinements. The column SP provides information whether preservation of some shape properties was verified (together with the default checks of undesired manipulation of null, undefined, or deleted objects).

Despite the prototype nature of our tool, which can still be optimised in many ways (some of them are mentioned below), the results are quite positive. For example, for checking the Deutsch–Schorr–Waite tree traversal, TVLA (version 2) took 1 minute on the same machine with manually provided instrumentation predicates and predicate transformers. In the case of the trees with linked leaves, we are not aware of any other fully automated tool with which experiments with this structure have been performed.

Recent fully automated tools based on separation logic appear to be more scalable, but the abstraction used in them is much more restricted to a particular shape of data structures (usually lists of lists [9,20,60] like in the case of task-lists[23])

[21] A precise (but more costly) spuriousness check is to replay the obtained path from the beginning without using abstraction.

[22] www.fit.vutbr.cz/research/groups/verifit/tools/artmc/

[23] In [31], more complex structures are automatically manipulated, but they are built by the programs in some "nice" way, suitable for the inductive predicates being constructed and used.

Table 3 Results of experiments with analysing programs manipulating dynamic data structures

Example	Time	Abstraction method	$\|Q\|$	N_{ref}	SP
Creation of SLLs	1 s	Predicates, restricted	25	0	Yes
Reversion of SLLs	5 s	Predicates	52	0	Yes
Deletion from DLLs	6 s	Finite height	100	0	Yes
Insertion into DLLs	10 s	Neighbour, restricted	106	0	Yes
Reversion of DLLs	7 s	Predicates	54	0	Yes
Insertsort of DLLs	2 s	Predicates	51	0	No
Inserting into trees	23 s	Predicates, restricted	65	0	Yes
Depth-first search	11 s	Predicates	67	1	Yes
Linking leaves in trees	40 s	Predicates	75	2	Yes
Inserting into a list of lists	5 s	Predicates, restricted	55	0	Yes
Deutsch–Schorr–Waite tree traversal	47 s	Predicates	126	0	No
Insertion into task-lists	11 m 25 s	Finite-height, restricted	277	0	Yes
Deletion in task-lists	1 m 41 s	Predicates, restricted	420	0	Yes

and their particular properties. For example, the abstraction of [60] is fine-tuned not to report spurious errors in the considered experiments and the authors themselves make a note on usefulness of automated refinement.

7 Conclusions

We have discussed the approach of abstract regular (tree) model checking as a generic technique for verification of parameterised and infinite-state programs, using finite word and tree automata for finitely representing possibly infinite sets of reachable configurations. As a possible application of ARTMC, we have discussed verification of programs manipulating complex dynamic data structures. Other applications include verification of parameterised networks of processes or systems with unbounded queues, stacks, counters, etc.

An important on-going research on (abstract) regular (tree) model checking includes improvements in the underlying automata technology. This especially concerns the use of non-deterministic word or tree automata. For them to be useful, one needs to be able to perform all the needed operations without determinising the automata. This is problematic in the case of inclusion checking and reduction (minimisation) of the automata. However, recent advances in using antichain-based and/or simulation-based approaches [1,26] have overcome this obstacle, although further improvements are still possible. Very significant improvements in the performance of ARTMC have already been reported on some case studies [11]. Further, development of abstractions and encodings particularly suitable for various application areas is needed. For instance, an interesting question is how to introduce the principle of separation (frame rule), which underlies the scalability of separation-logic-based approaches, into automata, combining it with automated refinement and higher generality while still retaining full automation [32].

Acknowledgments This work was supported by the Czech Science Foundation (projects P103/10/0306 and 201/09 /P531), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the internal BUT FIT grant FIT-S-11-1, the Czech-French Barrande project MEB021023, the French ANR RNTL Averiles project and the French ANR-09-SEGI Veridyc project.

References

1. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When simulation meets antichains (on Checking Language Inclusion of NFAs). In: Proceedings of TACAS'10. LNCS, vol. 6015. Springer, Berlin (2010)
2. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for MSO + LTL. In: Proceedings of CAV'04. LNCS, vol. 3114. Springer, Berlin (2004)
3. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for LTL(MSO). Special section on regular model checking. STTT (2010, this volume)
4. Abdulla, P.A., d'Orso, J., Jonsson, B., Nilsson, M.: Regular model checking made simple and efficient. In: Proceedings of CONCUR'02. LNCS, vol. 2421. Springer, Berlin (2002)
5. Abdulla, P.A., d'Orso, J., Jonsson, B., Nilsson, M.: Algorithmic improvements in regular model checking. In: Proceedings of CAV'03. LNCS, vol. 2725. Springer, Berlin (2003)
6. Abdulla, P.A., Jonsson, B., Mahata, P., d'Orso, J.: Regular tree model checking. In: Proceedings of CAV'02. LNCS, vol. 2404. Springer, Berlin (2002)
7. Abdulla, P.A., Legay, A., d'Orso, J., Rezine A.: Simulation-based iteration of tree transducers. In: Proceedings of TACAS'05. LNCS, vol. 3440. Springer, Berlin (2005)
8. Bensalem, S., Lakhnech, Y., Owre, S.: Computing abstractions of infinite state systems compositionally and automatically. In: Proceedings of CAV'98. LNCS, vol. 1427. Springer, Berlin (1998)
9. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures.

In: Proceedings of CAV'07. LNCS, vol. 4490. Springer, Berlin (2007)
10. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Proceedings of CAV'03. LNCS, vol. 2725. Springer, Berlin (2003)
11. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In: Proceedings of CIAA'08. LNCS, vol. 5148. Springer, Berlin (2008)
12. Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying programs with dynamic 1-selector-linked structures in regular model checking. In: Proceedings of TACAS'05. LNCS, vol. 3440. Springer, Berlin (2005)
13. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking. In: Proceedings of Infinity'05. ENTCS 149:37–48 (2006)
14. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Proceedings of SAS'06. LNCS, vol. 4134. Springer, Berlin (2006)
15. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Proceedings of CAV'04. LNCS, vol. 3114. Springer, Berlin (2004)
16. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Proceedings of CAV'00. LNCS, vol. 1855. Springer, Berlin (2000)
17. Bouajjani, A., Legay, A., Wolper, P.: Handling liveness properties in (ω-)regular model checking. In: Proceedings of Infinity'04. ENTCS 138:101–115 (2005)
18. Bouajjani, A., Touili, T.: Extrapolating tree transformations. In: Proceedings of CAV'02. LNCS, vol. 2404. Springer, Berlin (2002)
19. Bouajjani, A., Touili, T.: Widening techniques for regular tree model checking. Special section on regular model checking. STTT (2010, this volume)
20. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proceedings of POPL'09. ACM Press, New York (2009)
21. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of CAV'00. LNCS, vol. 1855. Springer, Berlin (2000)
22. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata (2005)
23. Dams, D., Lakhnech, Y., Steffen, M.: Iterating transducers. In: Proceedings of CAV'01. LNCS, vol. 2102. Springer, Berlin (2001)
24. Das, S., Dill, D.L.: Counter-example based predicate discovery in predicate abstraction. In: Proceedings of FMCAD'02 (2002)
25. Deshmukh, J.V., Emerson, E.A., Gupta, P.: Automatic verification of parameterized data structures. In: Proceedings of TACAS'06. LNCS, vol. 3920. Springer, Berlin (2006)
26. Doyen, L., Raskin, J.-F.: Antichain algorithms for finite automata. In: Proceedings of TACAS'10. LNCS, vol. 6015. Springer, Berlin (2010)
27. Engelfriet, J.: Bottom-up and top-down tree transformations—a comparison. Math. Syst. Theory **9**, 198–231 (1975)
28. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Proceedings of CAV'00. LNCS, vol. 1855. Springer, Berlin (2000)
29. Fribourg, L., Olsen, H.: Reachability sets of parametrized rings as regular languages. In: Proceedings of Infinity'97, ENTCS 9 (1997)
30. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Proceedings of CAV'97. LNCS, vol. 1254. Springer, Berlin (1997)
31. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: Proceedings of PLDI'07. ACM Press, New York (2007)
32. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: Proceedings of CAV'11. LNCS, vol. 6806. Springer, Berlin (2011)
33. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. In: Proceedings of Infinity'04. ENTCS 138:21–36 (2005)
34. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of POPL'02. ACM Press, New York (2002)
35. Jonsson, B., Nilsson, M.: Transitive closures of regular relations for verifying infinite-state systems. In: Proceedings of TACAS'00. LNCS, vol. 1785. Springer, Berlin (2000)
36. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. In: Proceedings of CAV'97. LNCS, vol. 1254. Springer, Berlin (1997)
37. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark (2001)
38. Klarlund, N., Schwartzbach, M.I.: Graph types. In: Proceedings of POPL'93. ACM Press, New York (1993)
39. Legay, A.: Extrapolating (Omega-)regular model checking. Special section on regular model checking. STTT (2010, this volume)
40. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proceedings of PLDI'01. ACM Press, New York (2001)
41. Nilsson, M.: Regular model checking. Licentiate Thesis, Uppsala University, Sweden (2000)
42. Nilsson, M.: Regular model checking. PhD thesis, Uppsala University (2005)
43. Perrin, D., Pin, J.-E.: Infinite Words: Automata, Semigroups, Logic and Games. Academic Press, New York (2003)
44. Pnueli, A., Shahar, E.: Liveness and acceleration in parameterized verification. In: Proceedings of CAV 2000. LNCS, vol. 1855. Springer, Berlin (2000)
45. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of LICS'02. IEEE CS Press (2002)
46. Rogalewicz, A.: Verification of programs with complex data structures. PhD thesis, FIT, Brno University of Technology (2005)
47. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS **24**(3), 1–50 (2002)
48. Saidi, H.: Model checking guided abstraction and analysis. In: Proceedings of SAS'00. LNCS, vol. 1824. Springer, Berlin (2000)
49. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. In: Proceedings of Infinity'05 (2005)
50. Shahar, E.: Tools and techniques for verifying parameterized systems. PhD thesis, Weizmann Institute of Science, Rehovot, Israel (2001)
51. Shahar, E., Pnueli, A.: Acceleration in verification of parameterized tree networks. Technical Report MCS02-12, Weizmann Institute of Science, Rehovot, Israel (2002)
52. Touili, T.: Regular model checking using widening techniques. ENTCS **50** (2001)
53. van Noord, G.: FSA6.2, 2004. http://odur.let.rug.nl/~vannoord/Fsa/
54. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Actively learning to verify safety for FIFO automata. In: Proceedings of FSTTCS'04. LNCS, vol. 3328. Springer, Berlin (2004)
55. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Learning to verify safety properties. In: Proceedings of ICFEM'04. LNCS, vol. 3308. Springer, Berlin (2004)
56. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Using language inference to verify omega-regular properties. In: Proceedings of TACAS'05. LNCS, vol. 3440. Springer, Berlin (2005)
57. Vardhan, A., Viswanathan, M.: Learning to verify branching time properties. In: Proceedings of ASE'05. IEEE/ACM (2005)
58. Vojnar, T.: Cut-offs and automata in formal verification of infinite-state systems. Habilitation thesis, FIT, Brno University of Technology, Czech Republic (2007)

59. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Proceedings of CAV'98. LNCS, vol. 1427. Springer, Berlin (1998)

60. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Proceedings of CAV'08. LNCS, vol. 5123. Springer, Berlin (2008)

Appendix D

The Tree Width of Separation Logic with Recursive Definitions

The Tree Width of Separation Logic with Recursive Definitions

Radu Iosif[1], Adam Rogalewicz[2], and Jiri Simacek[2]

[1] VERIMAG/CNRS, Grenoble, France
[2] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Separation Logic is a widely used formalism for describing dynamically allocated linked data structures, such as lists, trees, etc. The decidability status of various fragments of the logic constitutes a long standing open problem. Current results report on techniques to decide satisfiability and validity of entailments for Separation Logic(s) over lists (possibly with data). In this paper we establish a more general decidability result. We prove that any Separation Logic formula using rather general recursively defined predicates is decidable for satisfiability, and moreover, entailments between such formulae are decidable for validity. These predicates are general enough to define (doubly-) linked lists, trees, and structures more general than trees, such as trees whose leaves are chained in a list. The decidability proofs are by reduction to decidability of Monadic Second Order Logic on graphs with bounded tree width.

1 Introduction

Separation Logic (SL) [17] is a general framework for describing dynamically allocated mutable data structures generated by programs that use pointers and low-level memory allocation primitives. The logics in this framework are used by an important number of academic (SPACE INVADER [1], SLEEK [16] and PREDATOR [9]), as well as industrial-scale (INFER [7]) tools for program verification and certification. These logics are used both externally, as property specification languages, or internally, as e.g., abstract domains for computing invariants, or for proving verification conditions. The main advantage of using SL when dealing with heap manipulating programs, is the ability to provide compositional proofs, based on the principle of *local reasoning* i.e., analyzing different sections (e.g., functions, threads, etc.) of the program, that work on disjoint parts of the global heap, and combining the analysis results a-posteriori.

The basic language of SL consists of two kinds of atomic propositions describing either (i) the empty heap, or (ii) a heap consisting of an allocated cell, connected via a separating conjunction primitive. Hence a basic SL formula can describe only a heap whose size is bounded by the size of the formula. The ability of describing unbounded data structures is provided by the use of *recursive definitions*. Figure 1 gives several common examples of recursive data structures definable in this framework.

The main difficulty that arises when using Separation Logic with Recursive Definitions (SLRD) to reason automatically about programs is that the logic, due to its expressiveness, does not have very nice decidability properties. Most dialects used in practice restrict the language (e.g., no quantifier alternation, the negation is used in a

M.P. Bonacina (Ed.): CADE 2013, LNAI 7898, pp. 21–38, 2013.

$$
\begin{array}{rcl}
list(hd,tl) & ::= & emp \wedge hd = tl \\
& | & \exists x.\ hd \mapsto x * list(x,tl) \\
dll(hd,p,tl) & ::= & emp \wedge hd = tl \\
& | & \exists x.\ hd \mapsto (x,p) * dll(x,hd,tl) \\
tree(root) & ::= & emp \wedge root = nil \\
& | & \exists l,r.\ root \mapsto (l,r) * tree(l) * tree(r) \\
tll(x,ll,lr) & ::= & x \mapsto (nil,nil,lr) \wedge x = ll \\
& | & \exists l,r,z.\ x \mapsto (l,r,nil) * tll(l,ll,z) \\
& & * tll(r,z,lr)
\end{array}
$$

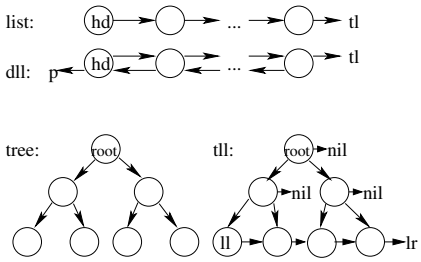


Fig. 1. Examples of recursive data structures definable in SLRD

very restricted ways, etc.) and the class of models over which the logic is interpreted (typically singly-linked lists, and slight variations thereof). In the same way, we apply several natural restrictions on the syntax of the recursive definitions, and define the fragment SLRD$_{btw}$, which guarantees that all models of a formula in the fragment have *bounded tree width*. Indeed, this ensures that the satisfiability and entailment problems in this fragment are decidable *without any restrictions on the type of the recursive data structures considered*.

In general, the techniques used in proving decidability of Separation Logic are either proof-based ([16,2]), or model-based ([5,8]). It is well-known that automata theory, through various automata-logics connections, provides a unifying framework for proving decidability of various logics, such as (W)SkS, Presburger Arithmetic or MSO over certain classes of graphs. In this paper we propose an automata-theoretic approach consisting of two ingredients. First, SLRD$_{btw}$ formulae are translated into equivalent Monadic Second Order (MSO) formulae over graphs. Second, we show that the models of SLRD$_{btw}$ formulae have the *bounded tree width* property, which provides a decidability result by reduction to the satisfiability problem for MSO interpreted over graphs of bounded tree width [18], and ultimately, to the emptiness problem of tree automata.

Related Work. The literature on defining decidable logics for describing mutable data structures is rather extensive. Initially, first-order logic with transitive closure of one function symbol was introduced in [11] with a follow-up logic of reachability on complex data structures, in [19]. The decision procedures for these logics are based on reductions to the decidability of MSO over finite trees. Along the same lines, the logic PALE [15] goes beyond trees, in defining trees with edges described by regular routing expressions, whose decidability is still a consequence of the decidability of MSO over trees. More recently, the CSL logic [4] uses first-order logic with reachability (along multiple selectors) in combination with arithmetic theories to reason about shape, path lengths and data within heap structures. Their decidability proof is based on a small model property, and the algorithm is enumerative. In the same spirit, the STRAND logic [14] combines MSO over graphs, with quantified data theories, and provides decidable fragments using a reduction to MSO over graphs of bounded tree width.

On what concerns SLRD [17], the first (proof-theoretic) decidability result on a restricted fragment defining only singly-linked lists was reported in [2], which describe a coNP algorithm. The full basic SL without recursive definitions, but with the magic

wand operator was found to be undecidable when interpreted *in any memory model* [6]. Recently, the entailment problem for SLRD over lists has been reduced to graph homomorphism in [8], and can be solved in PTIME. This method has been extended to reason nested and overlaid lists in [10]. The logic SLRD_{btw}, presented in this paper is, to the best of our knowledge, the first decidable SL that can define structures more general than lists and trees, such as e.g. trees with parent pointers and linked leaves.

2 Preliminaries

For a finite set S, we denote by $\|S\|$ its cardinality. We sometimes denote sets and sequences of variables as $\mathbf{x}$, the distinction being clear from the context. If $\mathbf{x}$ denotes a sequence, $(\mathbf{x})_i$ denotes its i-th element. For a partial function $f : A \rightharpoonup B$, and $\bot \notin B$, we denote $f(x) = \bot$ the fact that f is undefined at some point $x \in A$. By $f[a \leftarrow b]$ we denote the function λx . if $x = a$ then b else $f(x)$. The domain of f is denoted $dom(f) = \{x \in A \mid f(x) \neq \bot\}$, and the image of f is denoted as $img(f) = \{y \in B \mid \exists x \in A \,.\, f(x) = y\}$. By $f : A \rightharpoonup_{fin} B$ we denote any partial function whose domain is finite. Given two partial functions f, g defined on disjoint domains, we denote by $f \oplus g$ their union.

Stores, Heaps and States. We consider $PVar = \{u, v, w, \ldots\}$ to be a countable infinite set of *pointer variables* and $Loc = \{l, m, n, \ldots\}$ to be a countable infinite set of *memory locations*. Let $nil \in PVar$ be a designated variable, $null \in Loc$ be a designated location, and $Sel = \{1, \ldots, \mathcal{S}\}$, for some given $\mathcal{S} > 0$, be a finite set of natural numbers, called *selectors* in the following.

Definition 1. *A* state *is a pair $\langle s, h\rangle$ where $s : PVar \rightharpoonup Loc$ is a partial function mapping pointer variables into locations such that $s(nil) = null$, and $h : Loc \rightharpoonup_{fin} Sel \rightharpoonup_{fin} Loc$ is a finite partial function such that (i) $null \notin dom(h)$ and (ii) for all $\ell \in dom(h)$ there exist $k \in Sel$ such that $(h(\ell))(k) \neq \bot$.*

Given a state $S = \langle s, h\rangle$, s is called the *store* and h the *heap*. For any $k \in Sel$, we write $h_k(\ell)$ instead of $(h(\ell))(k)$, and $\ell \xrightarrow{k} \ell'$ for $h_k(\ell) = \ell'$. We sometimes call a triple $\ell \xrightarrow{k} \ell'$ an *edge*, and k is called a *selector*. Let $Img(h) = \bigcup_{\ell \in Loc} img(h(\ell))$ be the set of locations which are destinations of some selector edge in h. A location $\ell \in Loc$ is said to be *allocated* in $\langle s, h\rangle$ if $\ell \in dom(h)$ (i.e. it is the source of an edge), and *dangling* in $\langle s, h\rangle$ if $\ell \in [img(s) \cup Img(h)] \setminus dom(h)$, i.e., it is either referenced by a store variable, or reachable from an allocated location in the heap, but it is not allocated in the heap itself. The set $loc(S) = img(s) \cup dom(h) \cup Img(h)$ is the set of all locations either allocated or referenced in a state $S = \langle s, h\rangle$.

Trees. Let Σ be a finite label alphabet, and $\mathbb{N}^*$ be the set of sequences of natural numbers. Let $\varepsilon \in \mathbb{N}^*$ denote the empty sequence, and $p.q$ denote the concatenation of two sequences $p, q \in \mathbb{N}^*$. A *tree* t over Σ is a finite partial function $t : \mathbb{N}^* \rightharpoonup_{fin} \Sigma$, such that $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$, and for each $p \in dom(t)$ and $i \in \mathbb{N}$, we have: $t(p.i) \neq \bot \Rightarrow \forall 0 \leq j < i \,.\, t(p.j) \neq \bot$. Given two positions $p, q \in dom(t)$, we say

that q is the i-th successor (child) of p if $q = p.i$, for $i \in \mathbb{N}$. Also q is a successor of p, or equivalently, p is the parent of q, denoted $p = parent(q)$ if $q = p.i$, for some $i \in \mathbb{N}$.

We will sometimes denote by $\mathcal{D}(t) = \{-1, 0, \ldots, N\}$ the *direction alphabet* of t, where $N = \max\{i \in \mathbb{N} \mid p.i \in dom(t)\}$. The concatenation of positions is defined over $\mathcal{D}(t)$ with the convention that $p.(-1) = q$ if and only if $p = q.i$ for some $i \in \mathbb{N}$. We denote $\mathcal{D}_+(t) = \mathcal{D}(t) \setminus \{-1\}$. A *path* in t, from p_1 to p_k, is a sequence $p_1, p_2, \ldots, p_k \in dom(t)$ of pairwise distinct positions, such that either $p_i = parent(p_{i+1})$ or $p_{i+1} = parent(p_i)$, for all $1 \leq i < k$. Notice that a path in the tree can also link sibling nodes, not just ancestors to their descendants, or viceversa. However, a path may not visit the same tree position twice.

Tree Width. A state (Def. 1) can be seen as a directed graph, whose nodes are locations, and whose edges are defined by the selector relation. Some nodes are labeled by program variables (*PVar*) and all edges are labeled by selectors (*Sel*). The notion of tree width is then easily adapted from generic labeled graphs to states. Intuitively, the tree width of a state (graph) measures the similarity of the state to a tree.

Definition 2. *Let $S = \langle s, h \rangle$ be a state. A* tree decomposition *of S is a tree $t : \mathbb{N}^* \rightharpoonup_{fin} 2^{loc(S)}$, labeled with sets of locations from $loc(S)$, with the following properties:*

1. *$loc(S) = \bigcup_{p \in dom(t)} t(p)$, the tree covers the locations of S*
2. *for each edge $l_1 \xrightarrow{s} l_2$ in S, there exists $p \in dom(t)$ such that $l_1, l_2 \in t(p)$*
3. *for each $p, q, r \in dom(t)$, if q is on a path from p to r in t, then $t(p) \cap t(r) \subseteq t(q)$*

The width of the decomposition is $w(t) = \max_{p \in dom(t)}\{\|t(p)\| - 1\}$. The tree width of S is $tw(S) = \min\{w(t) \mid t \text{ is a tree decomposition of } S\}$.

A set of states is said to have *bounded tree width* if there exists a constant $k \geq 0$ such that $tw(S) \leq k$, for any state S in the set. Figure 2 gives an example of a graph (left) and a possible tree decomposition (right).

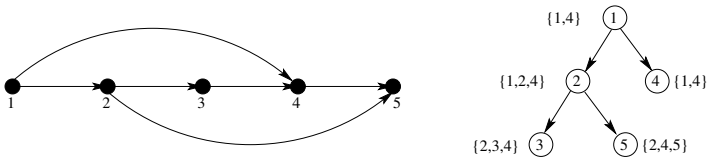


Fig. 2. A graph and a possible tree decomposition of width 2

2.1 Syntax and Semantics of Monadic Second Order Logic

Monadic second-order logic (MSO) on states is a straightforward adaptation of MSO on labeled graphs [13]. As usual, we denote first-order variables, ranging over locations, by $x, y, \ldots$, and second-order variables, ranging over sets of locations, by $X, Y, \ldots$. The set of logical MSO variables is denoted by $LVar_{mso}$, where $PVar \cap LVar_{mso} = \emptyset$.

We emphasize here the distinction between the logical variables $LVar_{mso}$ and the pointer variables $PVar$: the former may occur within the scope of first and second order quantifiers, whereas the latter play the role of symbolic constants (function symbols of zero arity). For the rest of this paper, a logical variable is said to be free if it does not occur within the scope of a quantifier. By writing $\varphi(\mathbf{x})$, for an MSO formula φ, and a set of logical variables $\mathbf{x}$, we mean that all free variables of φ are in $\mathbf{x}$.

The syntax of MSO is defined below:

$$u \in PVar;\ x, X \in LVar_{mso};\ k \in \mathbb{N}$$
$$\varphi ::= x = y \mid var_u(x) \mid edge_k(x,y) \mid null(x) \mid X(x) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists x.\varphi \mid \exists X.\varphi$$

The semantics of MSO on states is given by the relation $S, \iota, \nu \models_{mso} \varphi$, where $S = \langle s, h \rangle$ is a state, $\iota : \{x, y, z, \ldots\} \rightharpoonup_{fin} Loc$ is an interpretation of the first order variables, and $\nu : \{X, Y, Z, \ldots\} \rightharpoonup_{fin} 2^{Loc}$ is an interpretation of the second order variables. If $S, \iota, \nu \models_{mso} \varphi$ for all interpretations $\iota : \{x, y, z, \ldots\} \rightharpoonup_{fin} Loc$ and $\nu : \{X, Y, Z, \ldots\} \rightharpoonup_{fin} 2^{Loc}$, then we say that S is a *model* of φ, denoted $S \models_{mso} \varphi$. We use the standard MSO semantics [18], with the following interpretations of the vertex and edge labels:

$$\begin{array}{rcl} S, \iota, \nu \models_{mso} null(x) & \iff & \iota(x) = nil \\ S, \iota, \nu \models_{mso} var_u(x) & \iff & s(u) = \iota(x) \\ S, \iota, \nu \models_{mso} edge_k(x,y) & \iff & h_k(\iota(x)) = \iota(y) \end{array}$$

The *satisfiability problem* for MSO asks, given a formula φ, whether there exists a state S such that $S \models_{mso} \varphi$. This problem is, in general, undecidable. However, one can show its decidability on a restricted class of models. The theorem below is a slight variation of a classical result in (MSO-definable) graph theory [18]. For space reasons, all proofs are given in [12].

Theorem 1. *Let $k \geq 0$ be an integer constant, and φ be an MSO formula. The problem asking if there exists a state S such that $tw(S) \leq k$ and $S \models_{mso} \varphi$ is decidable.*

2.2 Syntax and Semantics of Separation Logic

Separation Logic (SL) [17] uses only a set of first order logical variables, denoted as $LVar_{sl}$, ranging over locations. We suppose that $LVar_{sl} \cap PVar = \emptyset$ and $LVar_{sl} \cap LVar_{mso} = \emptyset$. Let Var_{sl} denote the set $PVar \cup LVar_{sl}$. A formula is said to be *closed* if it does not contain logical variables which are not under the scope of a quantifier. By writing $\varphi(\mathbf{x})$ for an SL formula φ and a set of logical variables $\mathbf{x}$, we mean that all free variables of φ are in $\mathbf{x}$.

Basic Formulae. The syntax of basic formula is given below:

$$\alpha \in Var_{sl} \setminus \{nil\};\ \beta \in Var_{sl};\ x \in LVar_{sl}$$
$$\pi ::= \alpha = \beta \mid \alpha \neq \beta \mid \pi_1 \wedge \pi_2$$
$$\sigma ::= emp \mid \alpha \mapsto (\beta_1, \ldots, \beta_n) \mid \sigma_1 * \sigma_2 \text{ , for some } n > 0$$
$$\varphi ::= \pi \wedge \sigma \mid \exists x\ .\ \varphi$$

A formula of the form $\bigwedge_{i=1}^{n} \alpha_i = \beta_i \wedge \bigwedge_{j=1}^{m} \alpha_j \neq \beta_j$ defined by π in the syntax above is said to be *pure*. If Π is a pure formula, let Π^* denote its *closure*, i.e., the equivalent pure formula obtained by the exhaustive application of the reflexivity, symmetry, and transitivity axioms of equality. A formula of the form $\bigstar_{i=1}^{k} \alpha_i \mapsto (\beta_{i,1}, \ldots, \beta_{i,n})$ defined by σ in the syntax above is said to be *spatial*. The atomic proposition *emp* denotes the empty spatial conjunction. For a spatial formula Σ, let $|\Sigma|$ be the total number of variable occurrences in Σ, e.g. $|emp| = 0$, $|\alpha \mapsto (\beta_1, \ldots, \beta_n)| = n+1$, etc.

The semantics of a basic formula φ is given by the relation $S, \iota \models_{sl} \varphi$ where $S = \langle s, h \rangle$ is a state, and $\iota : LVar_{sl} \rightharpoonup_{fin} Loc$ is an interpretation of logical variables from φ. For a closed formula φ, we denote by $S \models_{sl} \varphi$ the fact that S is a *model* of φ.

$$
\begin{array}{ll}
S, \iota \models_{sl} emp & \iff dom(h) = \emptyset \\
S, \iota \models_{sl} \alpha \mapsto (\beta_1, \ldots, \beta_n) & \iff h = \{\langle (s \oplus \iota)(\alpha), \lambda i \,.\, \text{if } i \leq n \text{ then } (s \oplus \iota)(\beta_i) \text{ else } \bot \rangle\} \\
S, \iota \models_{sl} \varphi_1 * \varphi_2 & \iff S_1, \iota \models_{sl} \varphi_1 \text{ and } S_2, \iota \models_{sl} \varphi_2 \text{ where } S_1 \uplus S_2 = S
\end{array}
$$

The semantics of $=$, $\neq$, $\wedge$, and $\exists$ is classical. Here, the notation $S_1 \uplus S_2 = S$ means that S is the union of two states $S_1 = \langle s_1, h_1 \rangle$ and $S_2 = \langle s_2, h_2 \rangle$ whose stacks agree on the evaluation of common program variables ($\forall \alpha \in PVar \,.\, s_1(\alpha) \neq \bot \wedge s_2(\alpha) \neq \bot \Rightarrow s_1(\alpha) = s_2(\alpha)$), and whose heaps have disjoint domains ($dom(h_1) \cap dom(h_2) = \emptyset$) i.e., $S = \langle s_1 \cup s_2, h_1 \oplus h_2 \rangle$. Note that we adopt here the *strict semantics*, in which a points-to relation $\alpha \mapsto (\beta_1, \ldots, \beta_n)$ holds in a state consisting of a single cell pointed to by α, with exactly n outgoing edges towards dangling locations pointed to by $\beta_1, \ldots, \beta_n$, and the empty heap is specified by *emp*.

Every basic formula φ is equivalent to an existentially quantified pair $\Sigma \wedge \Pi$ where Σ is a spatial formula and Π is a pure formula. Given a basic formula φ, one can define its spatial (Σ) and pure (Π) parts uniquely, up to equivalence. A variable $\alpha \in Var$ is said to be *allocated in* φ if and only if $\alpha \mapsto (\ldots)$ occurs in Σ. It is easy to check that an allocated variable may not refer to a dangling location in any model of φ. A variable β is *referenced* if and only if $\alpha \mapsto (\ldots, \beta, \ldots)$ occurs in Σ for some variable α. For a basic formula $\varphi \equiv \Sigma \wedge \Pi$, the *size of* φ is defined as $|\varphi| = |\Sigma|$.

Lemma 1. *Let $\varphi(\mathbf{x})$ be a basic SL formula, $S = \langle s, h \rangle$ be a state, and $\iota : LVar_{sl} \rightharpoonup_{fin} Loc$ be an interpretation, such that $S, \iota \models_{sl} \varphi(\mathbf{x})$. Then $tw(S) \leq \max(|\varphi|, \|PVar\|)$.*

Recursive Definitions. A system $\mathcal{P}$ of *recursive definitions* is of the form:

$$
\begin{array}{c}
P_1(x_{1,1}, \ldots, x_{1,n_1}) ::= \mid_{j=1}^{m_1} R_{1,j}(x_{1,1}, \ldots, x_{1,n_1}) \\
\ldots \\
P_k(x_{k,1}, \ldots, x_{k,n_k}) ::= \mid_{j=1}^{m_k} R_{k,j}(x_{k,1}, \ldots, x_{k,n_k})
\end{array}
$$

where $P_1, \ldots, P_k$ are called *predicates*, $x_{i,1}, \ldots, x_{i,n_i}$ are called *parameters*, and the formulae $R_{i,j}$ are called the *rules* of P_i. Concretely, a rule $R_{i,j}$ is of the form $R_{i,j}(\mathbf{x}) \equiv \exists \mathbf{z} \,.\, \Sigma * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m) \wedge \Pi$, where Σ is a spatial SL formula over variables $\mathbf{x} \cup \mathbf{z}$, called the *head* of $R_{i,j}$, $\langle P_{i_1}(\mathbf{y}_1), \ldots, P_{i_m}(\mathbf{y}_m) \rangle$ is an ordered sequence of *predicate occurrences*, called the *tail* of $R_{i,j}$ (we assume w.l.o.g. that $\mathbf{x} \cap \mathbf{z} = \emptyset$, and that $\mathbf{y}_k \subseteq \mathbf{x} \cup \mathbf{z}$, for all $k = 1, \ldots, m$), Π is a pure formula over variables $\mathbf{x} \cup \mathbf{z}$.

Without losing generality, we assume that all variables occurring in a rule of a recursive definition system are logical variables from $LVar_{sl}$ – pointer variables can be passed as parameters at the top level. We subsequently denote $head(R_{i,j}) \equiv \Sigma$, $tail(R_{i,j}) \equiv \langle P_{i_k}(\mathbf{y}_k)\rangle_{k=1}^{m}$ and $pure(R_{i,j}) \equiv \Pi$, for each rule $R_{i,j}$. Rules with empty tail are called *base cases*. For each rule $R_{i,j}$ let $\|R_{i,j}\|^{var} = \|\mathbf{z}\| + \|\mathbf{x}\|$ be the number of variables, both existentially quantified and parameters, that occur in $R_{i,j}$. We denote by $\|\mathcal{P}\|^{var} = \max\{\|R_{i.j}\|^{var} \mid 1 \leq i \leq k,\ 1 \leq j \leq m_i\}$ the maximum such number, among all rules in $\mathcal{P}$. We also denote by $\mathcal{D}(\mathcal{P}) = \{-1, 0, \ldots, \max\{|tail(R_{i,j})| \mid 1 \leq i \leq k,\ 1 \leq j \leq m_i\} - 1\}$ the *direction alphabet* of $\mathcal{P}$.

Example. The predicate tll describes a data structure called a *tree with parent pointers and linked leaves* (see Fig. 3(b)). The data structure is composed of a binary tree in which each internal node points to left and right children, and also to its parent node. In addition, the leaves of the tree are kept in a singly-linked list, according to the order in which they appear on the frontier (left to right).

$$\begin{array}{rll} tll(x,p,leaf_l,leaf_r) ::= & x \mapsto (nil,nil,p,leaf_r) \wedge x = leaf_l & (R_1) \\ \mid & \exists l,r,z.\ x \mapsto (l,r,p,nil) * tll(l,x,leaf_l,z) * tll(r,x,z,leaf_r) & (R_2) \end{array}$$

The base case rule (R_1) allocates leaf nodes. The internal nodes of the tree are allocated by the rule (R_2), where the ttl predicate occurs twice, first for the left subtree, and second for the right subtree. □

Definition 3. *Given a system of recursive definitions* $\mathcal{P} = \left\{P_i ::= |_{j=1}^{m_i} R_{i,j}\right\}_{i=1}^{n}$, *an* unfolding tree of $\mathcal{P}$ rooted at i *is a finite tree* t *such that:*

1. *each node of* t *is labeled by a single rule of the system* $\mathcal{P}$,
2. *the root of* t *is labeled with a rule of* P_i,
3. *nodes labeled with base case rules have no successors, and*
4. *if a node* u *of* t *is labeled with a rule whose tail is* $P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m)$, *then the children of* u *form the ordered sequence* $v_1, \ldots, v_m$ *where* v_j *is labeled with one of the rules of* P_{i_j} *for all* $j = 1, \ldots, m$.

Remarks. Notice that the recursive predicate $P(x) ::= \exists y\ .\ x \mapsto y * P(y)$ does not have finite unfolding trees. However, in general a system of recursive predicates may have infinitely many finite unfolding trees. □

In the following, we denote by $\mathcal{T}_i(\mathcal{P})$ the set of unfolding trees of $\mathcal{P}$ rooted at i. An unfolding tree $t \in \mathcal{T}_i(\mathcal{P})$ corresponds to a basic formula of separation logic ϕ_t, called the *characteristic formula* of t, and defined in what follows. For a set of tree positions $P \subseteq \mathbb{N}^*$, we denote $LVar^P = \{x^p \mid x \in LVar,\ p \in P\}$. For a tree position $p \in \mathbb{N}^*$ and a rule R, we denote by R^p the rule obtained by replacing every variable occurrence x in R by x^p. For each position $p \in dom(t)$, we define a formula ϕ_t^p, by induction on the structure of the subtree of t rooted at p:

- if p is a leaf labeled with a base case rule R, then $\phi_t^p \equiv R^p$
- if p has successors $p.1, \ldots, p.m$, and the label of p is the recursive rule $R(\mathbf{x}) \equiv \exists \mathbf{z}\ .\ head(R) * \bigstar_{j=1}^{m} P_{i_j}(\mathbf{y}_j) \wedge pure(R)$, then:

$$\phi_t^p(\mathbf{x}^p) \equiv \exists \mathbf{z}^p\ .\ head(R^p) * \bigstar_{j=1}^{m} [\exists \mathbf{x}_{i_j}^{p.i}\ .\ \phi_t^{p.i}(\mathbf{x}_{i_j}^{p.i}) \wedge \mathbf{y}_j^p = \mathbf{x}_{i_j}^{p.i}] \wedge pure(R^p)$$

In the rest of the paper, we write ϕ_t for ϕ_t^{ε}. Notice that ϕ_t is defined using the set of logical variables $LVar^{dom(t)}$, instead of $LVar$. However the definition of SL semantics from the previous carries over naturally to this case.

Example. (cont'd) Fig. 3(a) presents an unfolding tree for the *tll* predicate given in the previous example. The characteristic formula of each node in the tree can be obtained by composing the formulae labeling the children of the node with the formula labeling the node. The characteristic formula of the tree is the formula of its root. □

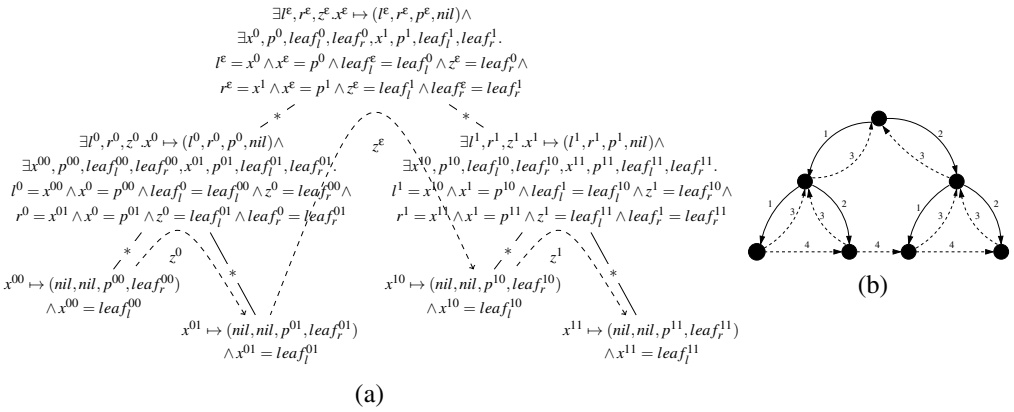


Fig. 3. (a) An unfolding tree for tll predicate and (b) a model of the corresponding formula

Given a system of recursive definitions $\mathcal{P} = \left\{P_i \ ::= \ |_{j=1}^{m_i} \ R_{i,j}\right\}_{i=1}^{n}$, the semantics of a recursive predicate P_i is defined as follows:

$$S,\iota \models_{sl} P_i(x_{i,1},\ldots,x_{i,n_i}) \iff S,\iota^{\varepsilon} \models_{sl} \phi_t(x_{i,1}^{\varepsilon},\ldots,x_{i,n_i}^{\varepsilon}), \text{ for some } t \in \mathcal{T}_i(\mathcal{P}) \qquad (1)$$

where $\iota^{\varepsilon}(x_{i,j}^{\varepsilon}) \stackrel{def}{=} \iota(x_{i,j})$ for all $j = 1,\ldots,n_i$.

Remark. Since the recursive predicate $P(x) ::= \exists y \ . \ x \mapsto y * P(y)$ does not have finite unfolding trees, the formula $\exists x.P(x)$ is unsatisfiable. □

Top Level Formulae. We are now ready to introduce the fragment of *Separation Logic with Recursive Definitions* (SLRD). A formula in this fragment is an existentially quantified formula of the following form: $\exists \mathbf{z} \ . \ \varphi * P_{i_1} * \ldots * P_{i_n}$, where φ is a basic formula, and P_{i_j} are occurrences of recursive predicates, with free variables in $PVar \cup \mathbf{z}$. The semantics of an SLRD formula is defined in the obvious way, from the semantics of the basic fragment, and that of the recursive predicates.

Example. The following SLRD formulae, with $PVar = \{root, head\}$, describe both the set of binary trees with parent pointer and linked leaves, rooted at *root*, with the leaves

linked into a list pointed to by *head*. The difference is that φ_1 describes also a tree containing only a single allocated location:

$\varphi_1 \equiv tll(root, nil, head, nil)$
$\varphi_2 \equiv \exists l, r, x. root \mapsto (l, r, nil, nil) * tll(l, root, head, x) * tll(r, root, x, nil)$ □

We are interested in solving two problems on SLRD formulae, namely *satisfiability* and *entailment*. The satisfiability problem asks, given a closed SLRD formula φ, whether there exists a state S such that $S \models_{sl} \varphi$. The entailment problem asks, given two closed SLRD formulae φ_1 and φ_2, whether for all states S, $S \models_{sl} \varphi_1$ implies $S \models_{sl} \varphi_2$. This is denoted also as $\varphi_1 \models_{sl} \varphi_2$. For instance, in the previous example we have $\varphi_2 \models_{sl} \varphi_1$, but not $\varphi_1 \models_{sl} \varphi_2$.

In general, it is possible to reduce an entailment problem $\varphi_1 \models \varphi_2$ to satisfiability of the formula $\varphi_1 \wedge \neg\varphi_2$. In our case, however, this is not possible directly, because SLRD is not closed under negation. The decision procedures for satisfiability and entailment is the subject of the rest of this paper.

3 Decidability of Satisfiability and Entailment in SLRD

The decision procedure for the satisfiability and entailment in SLRD is based on two ingredients. First, we show that, under certain natural restrictions on the system of recursive predicates, which define a fragment of SLRD, called SLRD_{btw}, all states that are models of SLRD_{btw} formulae have *bounded tree width* (Def. 2). These restrictions are as follows:

1. *Progress*: each rule allocates exactly one variable
2. *Connectivity*: there is at least one selector edge between the variable allocated by a rule and the variable allocated by each of its children in the unfolding tree
3. *Establishment*: all existentially quantified variables in a recursive rule are eventually allocated

Second, we provide a *translation of SLRD_{btw} formulae into equivalent MSO formulae*, and rely on the fact that satisfiability of MSO is decidable on classes of states with bounded tree width.

3.1 A Decidable Subset of SLRD

At this point we define the SLRD_{btw} fragment formally, by defining the three restrictions above. The *progress* condition (1) asks that, for each rule R in the system of recursive definitions, we have $head(R) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_n)$, for some variables $\alpha, \beta_1, \ldots, \beta_n \in Var_{sl}$. The intuition between this restriction is reflected by the following example.

Example. Consider the following system of recursive definitions:

$$ls(x,y) ::= x \mapsto y \mid \exists z,t \;.\; x \mapsto (z, nil) * t \mapsto (nil, y) * ls(z,t)$$

The predicate $ls(x,y)$ defines the set of structures $\{x(\xrightarrow{1})^n z \mapsto t(\xrightarrow{2})^n y \mid n \geq 0\}$, which clearly cannot be defined in MSO. □

The *connectivity* condition (2) is defined below:

Definition 4. *A rule R of a system of recursive definitions, such that $head(R) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_n)$ and $tail(R) \equiv \langle P_{i_1}(\mathbf{y}_1), \ldots, P_{i_m}(\mathbf{y}_m) \rangle$, $m \geq 1$, is said to be* connected *if and only if the following hold:*

- *for each $j = 1, \ldots, m$, $(\mathbf{y}_j)_s = \beta'$, for some $1 \leq s \leq n_{i_j}$, where n_{i_j} is the number of parameters of P_{i_j}*
- *$\beta_t = \beta'$ occurs in $pure(R)^*$, for some $1 \leq t \leq n$*
- *the s-th parameter $x_{i_j,s}$ of P_{i_j} is allocated in the heads of all rules of P_{i_j}.*

In this case we say that between rule R and any rule Q of P_{i_j}, there is a *local edge*, labeled by *selector* t. $\mathcal{F}(R, j, Q) \subseteq Sel$ denotes the set of all such selectors. If all rules of $\mathcal{P}$ are connected, we say that $\mathcal{P}$ is connected.

Example. The following recursive rule, from the previous tll predicate, is connected:

$$\exists l, r, z \;.\; x \mapsto (l, r, p, nil) * tll(l, x, leaf_l, z) * tll(r, x, z, leaf_r) \quad (R_2)$$

R_2 is connected because the variable l is referenced in R_2 and it is passed as the first parameter to tll in the first recursive call to tll. Moreover, the first parameter (x) is allocated by all rules of tll. R_2 is connected, for similar reasons. We have $\mathcal{F}(R_2, 1, R_2) = \{1\}$ and $\mathcal{F}(R_2, 2, R_2) = \{2\}$. □

The *establishment* condition (3) is formally defined below.

Definition 5. *Let $P(x_1, \ldots, x_n) = |_{j=1}^{m} R_j(x_1, \ldots, x_n)$ be a predicate in a recursive system of definitions. We say that a parameter x_i, for some $i = 1, \ldots, n$ is* allocated *in P if and only if, for all $j = 1, \ldots, m$:*

- *either x_i is allocated in $head(R_j)$, or*
- *(i) $tail(R_j) = \langle P_{i_1}(\mathbf{y}_1), \ldots, P_{i_k}(\mathbf{y}_k) \rangle$, (ii) $(\mathbf{y}_\ell)_s = x_i$ occurs in $pure(R_j)^*$, for some $\ell = 1, \ldots, k$, and (iii) the s-th parameter of P_{i_ℓ} is allocated in P_{i_ℓ}*

A system of recursive definitions is said to be established *if and only if every existentially quantified variable is allocated.*

Example. Let $llextra(x) ::= x \mapsto (nil, nil) \mid \exists n, e.\; x \mapsto (n, e) * llextra(n)$ be a recursive definition system, and let $\phi ::= llextra(head)$, where $head \in PVar$. The models of the formula ϕ are singly-linked lists, where in all locations of the heap, the first selector points to the next location in the list, and the second selector is dangling i.e., it can point to any location in the heap. These dangling selectors may form a squared grid of arbitrary size, which is a model of the formula ϕ. However, the set of squared grids does not have bounded tree width [18]. The problem arises due to the existentially quantified variables e which are never allocated. □

Given a system $\mathcal{P}$ of recursive definitions, one can effectively check whether it is established, by guessing, for each predicate $P_i(x_{i,1}, \ldots, x_{i,n_i})$ of $\mathcal{P}$, the minimal set of parameters which are allocated in P_i, and verify this guess inductively[1]. Then, once the minimal set of allocated parameters is determined for each predicate, one can check whether every existentially quantified variable is eventually allocated.

[1] For efficiency, a least fixpoint iteration can be used instead of a non-deterministic guess.

Lemma 2. *Let $\mathcal{P} = \{P_i ::= |_{j=1}^{m_i} R_{ij}(x_{i,1}, \ldots, x_{i,n_i})\}_{i=1}^{k}$ be a established system of recursive definitions, and $S = \langle s, h \rangle$ be a state, such that $S, \iota \models_{sl} P_i(x_{i,1}, \ldots, x_{i,n_i})$ for some interpretation $\iota : LVar_{sl} \rightharpoonup_{fin} Loc$ and some $1 \leq i \leq k$. Then $tw(S) \leq \|\mathcal{P}\|^{var}$.*

The result of the previous lemma extends to an arbitrary top-level formula:

Theorem 2. *Let $\mathcal{P} = \{P_i ::= |_{j=1}^{m_i} R_{ij}(x_{i,1}, \ldots, x_{i,n_i})\}_{i=1}^{k}$ be a established system of recursive definitions, and $S = \langle s, h \rangle$ be a state, such that $S \models_{sl} \exists \mathbf{z} \,.\, \varphi(\mathbf{y}_0) * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_n}(\mathbf{y}_n)$, where φ is a basic SL formula, and P_{i_j} are predicates of $\mathcal{P}$, and $\mathbf{y}_i \subseteq \mathbf{z}$, for all $i = 0, 1, \ldots, n$. Then $tw(S) \leq \max(\|\mathbf{z}\|, |\varphi|, \|PVar\|, \|\mathcal{P}\|^{var})$.*

4 From SLRD$_{btw}$ to MSO

This section describes the translation of a SL formula using recursively defined predicates into an MSO formula. We denote by $\Pi(X_0, \ldots, X_i, X)$ the fact that $X_0, \ldots, X_i$ is a partition of X, and by $\Sigma(x, X)$ the fact that X is a singleton with x as the only element.

4.1 Converting Basic SL Formulae to MSO

For every SL logical variable $x \in LVar_{sl}$ we assume the existence of an MSO logical variable $\overline{x} \in LVar_{mso}$, which is used to replace x in the translation. For every program variable $u \in PVar \setminus \{nil\}$ we assume the existence of a logical variable $\overline{x_u} \in LVar_{mso}$. The special variable $nil \in LVar_{sl}$ is translated into $\overline{x_{nil}} \in LVar_{mso}$ (with the associated MSO constraint $null(\overline{x_{nil}})$). In general, for any pointer or logical variable $\alpha \in Var_{sl}$, we denote by $\overline{\alpha}$, the logical MSO variable corresponding to it.

The translation of a pure SL formula $\alpha = \beta$, $\alpha \neq \beta$, $\pi_1 \wedge \pi_2$ is $\overline{\alpha} = \overline{\beta}$, $\neg(\overline{\alpha} = \overline{\beta})$, $\overline{\pi_1} \wedge \overline{\pi_2}$, respectively, where $\overline{\pi}(\overline{\alpha_1}, \ldots, \overline{\alpha_k})$ is the translation of $\pi(\alpha_1, \ldots, \alpha_k)$. Spatial SL formulae $\sigma(\alpha_1, \ldots, \alpha_k)$ are translated into MSO formulae $\overline{\sigma}(\overline{\alpha_1}, \ldots, \overline{\alpha_k}, X)$, where X is used for the set of locations allocated in σ. The fact that X actually denotes the domain of the heap, is ensured by the following MSO constraint:

$$Heap(X) \equiv \forall x \bigvee_{i=1}^{\|Sel\|} (\exists y \,.\, edge_i(x, y)) \leftrightarrow X(x)$$

The translation of basic spatial formulae is defined by induction on their structure:

$$\begin{array}{ll} \overline{emp}(X) & \equiv \forall x \,.\, \neg X(x) \\ \overline{(\alpha \mapsto (\beta_1, \ldots, \beta_n))}(X) & \equiv \Sigma(\overline{\alpha}, X) \wedge \bigwedge_{i=1}^{n} edge_i(\overline{\alpha}, \overline{\beta_i}) \wedge \bigwedge_{i=n+1}^{\|Sel\|} \forall x \,.\, \neg edge_i(\overline{\alpha}, x) \\ \overline{(\sigma_1 * \sigma_2)}(X) & \equiv \exists Y \exists Z \,.\, \overline{\sigma_1}(Y) \wedge \overline{\sigma_2}(Z) \wedge \Pi(Y, Z, X) \end{array}$$

The translation of a closed basic SL formula φ in MSO is defined as $\exists X \,.\, \overline{\varphi}(X)$, where $\overline{\varphi}(X)$ is defined as $\overline{(\pi \wedge \sigma)}(X) \equiv \overline{\pi} \wedge \overline{\sigma}(X)$, and $\overline{(\exists x \,.\, \varphi_1)}(X) \equiv \exists \overline{x} \,.\, \overline{\varphi_1}(X)$. The following lemma proves that the MSO translation of a basic SL formula defines the same set of models as the original SL formula.

Lemma 3. *For any state $S = \langle s,h\rangle$, any interpretation $\iota : LVar_{sl} \rightharpoonup_{fin} Loc$, and any basic SL formula φ, we have $S,\iota \models_{sl} \varphi$ if and only if $S,\overline{\iota},\nu[X \leftarrow dom(h)] \models_{mso} \overline{\varphi}(X) \wedge Heap(X)$, where $\overline{\iota} : LVar_{mso} \rightharpoonup_{fin} Loc$ is an interpretation of first order variables, such that $\overline{\iota}(x_u) = s(u)$, for all $u \in PVar$, and $\overline{\iota}(\overline{x}) = \iota(x)$, for all $x \in LVar_{sl}$, and $\nu : LVar_{mso} \rightharpoonup_{fin} 2^{Loc}$ is any interpretation of second-order variables.*

4.2 States and Backbones

The rest of this section is concerned with the MSO definition of states that are models of recursive SL formulae, i.e. formulae involving recursively defined predicates. The main idea behind this encoding is that any part of a state which is the model of a recursive predicate can be decomposed into a tree-like structure, called the *backbone*, and a set of edges between the nodes in this tree. Intuitively, the backbone is a spanning tree that uses only *local edges*. For instance, in the state depicted in Fig. 3(b), the local edges are drawn in solid lines.

Let $P_k(x_1,\ldots,x_n)$ be a recursively defined predicate of a system $\mathcal{P}$, and $S,\iota \models_{sl} P_k(x_1,\ldots,x_n)$, for some state $S = \langle s,h\rangle$ and some interpretation $\iota : LVar_{sl} \to Loc$. Then $S,\iota \models_{sl} \phi_t$, where $t \in \mathcal{T}_k(\mathcal{P})$ is an unfolding tree, ϕ_t is its characteristic formula, and $\mu : dom(t) \to dom(h)$ is the bijective tree that describes the allocation of nodes in the heap by rules labeling the unfolding tree. Recall that the direction alphabet of the system $\mathcal{P}$ is $\mathcal{D}(\mathcal{P}) = \{-1,0,\ldots,N-1\}$, where N is the maximum number of predicate occurrences within some rule of $\mathcal{P}$, and denote $\mathcal{D}_+(\mathcal{P}) = \mathcal{D}(\mathcal{P}) \setminus \{-1\}$. For each rule R_{ij} in $\mathcal{P}$ and each direction $d \in \mathcal{D}(\mathcal{P})$, we introduce a second order variable X_{ij}^d to denote the set of locations ℓ such that (i) $t(\mu^{-1}(\ell)) \equiv R_{ij}$ and (ii) $\mu^{-1}(\ell)$ is a d-th child, if $d \geq 0$, or $\mu^{-1}(\ell)$ is the root of t, if $d = -1$. Let $\overrightarrow{\mathbf{X}}$ be the sequence of X_{ij}^k variables, enumerated in some order. We use the following shorthands:

$$X_{ij}(x) \equiv \bigvee_{k \in \mathcal{D}(\mathcal{P})} X_{ij}^k(x) \qquad X_i(x) \equiv \bigvee_{1 \leq j \leq m_i} X_{ij}(x) \qquad X_i^k(x) \equiv \bigvee_{1 \leq j \leq m_i} X_{ij}^k(x)$$

to denote, respectively, locations that are allocated by a rule R_{ij} (X_{ij}), by a recursive predicate P_i (X_i), or by a predicate P_i, who are mapped to a k-th child (or to the root, if $k = -1$) in the unfolding tree of $\mathcal{P}$, rooted at i (X_i^k).

In order to characterize the backbone of a state, one must first define the local edges:

$$local\_edge_{i,j,p,q}^d(x,y) \equiv \bigwedge_{s \in \mathcal{F}(R_{i,j},d,R_{pq})} edge_s(x,y)$$

for all $d \in \mathcal{D}_+(\mathcal{P})$. Here $\mathcal{F}(R_{ij},d,R_{pq})$ is the set of forward local selectors for direction d, which was defined previously – notice that the set of local edges depends on the source and destination rules R_{ij} and R_{pq}, that label the corresponding nodes in the unfolding tree, respectively. The following predicate ensures that these labels are used correctly, and define the successor functions in the unfolding tree:

$$succ_d(x,y,\overrightarrow{\mathbf{X}}) \equiv \bigvee_{\substack{1 \leq i,p \leq M \\ 1 \leq j \leq m_i \\ 1 \leq q \leq m_p}} X_{ij}(x) \wedge X_{pq}^k(y) \wedge local\_edge_{i,j,p,q}^d(x,y)$$

for all $d \in \mathcal{D}_+(\mathcal{P})$. The definition of the backbone of a recursive predicate P_i in MSO follows tightly the definition of the unfolding tree of $\mathcal{P}$ rooted at i (Def. 3):

$$backbone_i(r, \overrightarrow{\mathbf{X}}, T) \equiv tree(r, \overrightarrow{\mathbf{X}}, T) \wedge X_i^{-1}(r) \wedge succ\_labels(\overrightarrow{\mathbf{X}})$$

where $tree(r, \overrightarrow{\mathbf{X}}, T)$ defines a tree[2] with domain T, rooted at r, with successor functions defined by $succ_0, \ldots, succ_{N-1}$, and $succ\_labels$ ensures that the labeling of each tree position (with rules of $\mathcal{P}$) is consistent with the definition of $\mathcal{P}$:

$$succ\_labels(\overrightarrow{\mathbf{X}}) \equiv \bigwedge_{\substack{1 \leq i \leq M \\ 1 \leq j \leq m_i}} X_{ij}(x) \rightarrow \bigwedge_{d=0}^{r_{ij}-1} \exists y \,.\, X_{k_d}^d(y) \wedge succ_d(x, y, \overrightarrow{\mathbf{X}}) \wedge \forall y \,.\, \bigwedge_{p=s_{ij}+1}^{\|Sel\|} \neg edge_p(x, y)$$

where we suppose that, for each rule R_{ij} of $\mathcal{P}$, we have $head(R_{ij}) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_{s_{ij}})$ and $tail(R_{ij}) = \langle P_{k_1}, \ldots, P_{k_{r_{ij}}} \rangle$, for some $r_{ij} \geq 0$, and some indexing $k_1, \ldots, k_{r_{ij}}$ of predicate occurrences within R_{ij}. The last conjunct ensures that a location allocated in R_{ij} does not have more outgoing edges than specified by $head(R_{ij})$. This condition is needed, since, unlike SL, the semantics of MSO does not impose strictness conditions on the number of outgoing edges.

4.3 Inner Edges

An edge between two locations is said to be *inner* if both locations are allocated in the heap. Let μ be the bijective tree defined in Sec. 4.2. The existence of an edge $\ell \xrightarrow{k} \ell'$ in S, between two arbitrary locations $\ell, \ell' \in dom(h)$, is the consequence of:

1. a basic points-to formula $\alpha \mapsto (\beta_1, \ldots, \beta_k, \ldots, \beta_n)$ that occurs in $\mu(\ell)$
2. a basic points-to formula $\gamma \mapsto (\ldots)$ that occurs in $\mu(\ell')$
3. a path $\mu(\ell) = p_1, p_2, \ldots, p_{m-1}, p_m = \mu(\ell')$ in t, such that the equalities $\beta_k^{p_1} = \delta_2^{p_2} = \ldots = \delta_{m-1}^{p_{m-1}} = \gamma^{p_m}$ are all logical consequences of ϕ_t, for some tree positions $p_2, \ldots, p_{m-1} \in dom(t)$ and some variables $\delta_2, \ldots, \delta_{m-1} \in LVar_{sl}$.

Notice that the above conditions hold only for inner edges. The (corner) case of edges leading to dangling locations is dealt with in [12].

Example. The existence of the edge from tree position 00 to 01 in Fig. 3(b), is a consequence of the following: (1) $x^{00} \mapsto (nil, nil, p^{00}, leaf_r^{00})$, (2) $x^{01} \mapsto (nil, nil, p^{01}, leaf_r^{01})$, and (3) $leaf_r^{00} = z^0 = leaf_l^{01} = x^{01}$. The reason for other dashed edges is similar. □

The main idea here is to encode in MSO the existence of such paths, in the unfolding tree, between the source and the destination of an edge, and use this encoding to define the edges. To this end, we use a special class of tree automata, called *tree-walking automata* (TWA) to recognize paths corresponding to sequences of equalities occurring within characteristic formulae of unfolding trees.

[2] For space reasons this definition can be found in [12].

Tree Walking Automata. Given a set of tree directions $\mathcal{D} = \{-1, 0, \ldots, N\}$ for some $N \geq 0$, a tree-walking automaton[3], is a tuple $A = (\Sigma, Q, q_i, q_f, \Delta)$ where Σ is a set of tree node labels, Q is a set of states, $q_i, q_f \in Q$ are the initial and final states, and $\Delta : Q \times (\Sigma \cup \{root\}) \times (\Sigma \cup \{?\}) \to 2^{Q \times (\mathcal{D} \cup \{\varepsilon\})}$ is the (non-deterministic) transition function. A configuration of A is a pair $\langle p, q \rangle$, where $p \in \mathcal{D}^*$ is a tree position, and $q \in Q$ is a state. A run of A over a Σ-labeled tree t is a sequence of configurations $\langle p_1, q_1 \rangle, \ldots, \langle p_n, q_n \rangle$, with $p_1, \ldots, p_n \in dom(t)$, such that for all $i = 1, \ldots, n-1$, we have $p_{i+1} = p_i.k$, where either:

1. $p_i \neq \varepsilon$ and $(q_{i+1}, k) \in \Delta(q_i, t(p_i), t(p_i.(-1)))$, for $k \in \mathcal{D} \cup \{\varepsilon\}$
2. $p_i = \varepsilon$ and $(q_{i+1}, k) \in \Delta(q_i, \sigma, ?)$, for $\sigma \in \{t(p_i) \cup root\}$ and $k \in \mathcal{D} \cup \{\varepsilon\}$

The run is said to be *accepting* if $q_1 = q_i$, $p_1 = \varepsilon$ and $q_n = q_f$.

Routing Automata. For a system of recursive definitions $\mathcal{P} = \left\{ P_i(x_{i,1}, \ldots, x_{i,n_i}) ::= \left|_{j=1}^{m_i} R_{ij}(x_{i,1}, \ldots, x_{i,n_i}) \right\}_{i=1}^{k}$, we define the TWA $A_{\mathcal{P}} = (\Sigma_{\mathcal{P}}, Q_{\mathcal{P}}, q_i, q_f, \Delta_{\mathcal{P}})$, where $\Sigma_{\mathcal{P}} = \{R_{ij}^k \mid 1 \leq i \leq k,\ 1 \leq j \leq m_i,\ k \in \mathcal{D}(\mathcal{P})\}$, $Q_{\mathcal{P}} = \{q_x^{var} \mid x \in LVar_{sl}\} \cup \{q_s^{sel} \mid s \in Sel\} \cup \{q_i, q_f\}$. The transition function $\Delta_{\mathcal{P}}$ is defined as follows:

1. $(q_i, k), (q_s^{sel}, \varepsilon) \in \Delta(q_i, \sigma, \tau)$ for all $k \in \mathcal{D}_+(\mathcal{P})$, all $s \in Sel$ and all $\sigma \in \Sigma_{\mathcal{P}} \cup \{root\}$, $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ i.e., the automaton first moves downwards chosing random directions, while in q_i, then changes to q_s^{sel} for some non-deterministically chosen selector s.
2. $(q_{\beta_s}^{var}, \varepsilon) \in \Delta(q_s^{sel}, R_{ij}^k, \tau)$ and $(q_f, \varepsilon) \in \Delta(q_\alpha^{var}, R_{ij}^k, \tau)$ for all $k \in \mathcal{D}(\mathcal{P})$ and $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ if and only if $head(R_{ij}) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_s, \ldots, \beta_m)$, for some $m > 0$ i.e., when in q_s^{sel}, the automaton starts tracking the destination β_s of the selector s through the tree. The automaton enters the final state when the tracked variable α is allocated.
3. for all $k \in \mathcal{D}_+(\mathcal{P})$, all $\ell \in \mathcal{D}(\mathcal{P})$ and all rules $R_{\ell q}$ of $P_\ell(x_{\ell,1}, \ldots, x_{\ell,n_\ell})$, we have $(q_{x_{\ell,j}}^{var}, k) \in \Delta(q_{y_j}^{var}, R_{ij}^l, \tau)$, for all $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$, and $(q_{y_j}^{var}, -1) \in \Delta(q_{x_{\ell,j}}^{var}, R_{\ell q}^k, R_{ij}^l)$ if and only if $tail(R_{ij})_k \equiv P_\ell(y_1, \ldots, y_{n_\ell})$ i.e., the automaton moves down along the k-th direction tracking $x_{\ell,j}$ instead of y_j, when the predicate $P_\ell(\mathbf{y})$ occurs on the k-th position in R_{ij}. Symmetrically, the automaton can also move up tracking y_j instead of $x_{\ell,j}$, in the same conditions.
4. $(q_\beta^{var}, \varepsilon) \in \Delta(q_\alpha^{var}, R_{ij}^k, \tau)$ for all $k \in \mathcal{D}(\mathcal{P})$ and all $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ if and only if $\alpha = \beta$ occurs in $pure(R_{ij})$ i.e., the automaton switches from tracking α to tracking β when the equality between the two variables occurs in R_{ij}, while keeping the same position in the tree.

The following lemma formalizes the correctness of the TWA construction:

Lemma 4. *Given a system of recursive definitions $\mathcal{P}$, and an unfolding tree $t \in \mathcal{T}_i(\mathcal{P})$ of $\mathcal{P}$, rooted at i, for any $x, y \in LVar_{sl}$ and $p, r \in dom(t)$, we have $\models_{sl} \phi_t \to x^p = y^r$ if and only if $A_{\mathcal{P}}$ has a run from $\langle p, q_x^{var} \rangle$ to $\langle r, q_y^{var} \rangle$ over t, where ϕ_t is the characteristic formula of t.*

[3] This notion of tree-walking automaton is a slightly modified but equivalent to the one in [3]. We give the translation of TWA into the original definition in [12].

To the routing automaton $A_{\mathcal{P}}$ corresponds the MSO formula $\Phi_{A_{\mathcal{P}}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}})$, where r maps to the root of the unfolding tree, $\overrightarrow{\mathbf{X}}$ is the sequence of second order variables X_{ij}^k defined previously, T maps to the domain of the tree, and $\overrightarrow{\mathbf{Y}}$ is a sequence of second-order variables X_q, one for each state $q \in Q_{\mathcal{P}}$. We denote by Y_s^{sel} and Y_f the variables from $\overrightarrow{\mathbf{Y}}$ that correspond to the states q_S^{sel} and q_f, for all $s \in Sel$, respectively. For space reasons, the definition of $\Phi_{A_{\mathcal{P}}}$ is given in [12]. With this notation, we define:

$$inner\_edges(r, \overrightarrow{\mathbf{X}}, T) \equiv \forall x \forall y \bigwedge_{s \in Sel} \exists \overrightarrow{\mathbf{Y}} \,.\, \Phi_{A_{\mathcal{P}}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}}) \wedge Y_s^{sel}(x) \wedge Y_f(y) \rightarrow edge_s(x,y)$$

4.4 Double Allocation

In order to translate the definition of a recursively defined SL predicate $P(x_1, \ldots, x_n)$ into an MSO formula $\overline{P}$, that captures the models of P, we need to introduce a sanity condition, imposing that recursive predicates which establish equalities between variables allocated at different positions in the unfolding tree, are unsatisfiable, due to the semantics of the separating conjunction of SL, which implicitly conjoins all local formulae of an unfolding tree. A double allocation occurs in the unfolding tree t if and only if there exist two distinct positions $p, q \in dom(t)$ and:

1. a basic points-to formula $\alpha \mapsto (\ldots)$ occurring in $t(p)$
2. a basic points-to formula $\beta \mapsto (\ldots)$ occurring in $t(q)$
3. a path $p = p_1, \ldots, p_m = q$ in t, such that the equalities $\alpha^p = \gamma_2^{p_2} = \ldots = \gamma_{m-1}^{p_{m-1}} = \beta^q$ are all logical consequences of ϕ_t, for some tree positions $p_2, \ldots, p_{m-1} \in dom(t)$ and some variables $\gamma_2, \ldots, \gamma_{m-1} \in LVar_{sl}$

The cases of double allocation can be recognized using a routing automaton $B_{\mathcal{P}} = (\Sigma_{\mathcal{P}}, Q'_{\mathcal{P}}, q_i, q_f, \Delta'_{\mathcal{P}})$, whose states $Q'_{\mathcal{P}} = \{q_x^{var} \mid x \in LVar_{sl}\} \cup \{q_0, q_i, q_f\}$ and transitions $\Delta'_{\mathcal{P}}$ differ from $A_{\mathcal{P}}$ only in the following rules:

- $(q_0, \varepsilon) \in \Delta(q_i, \sigma, \tau)$ for all $\sigma \in \Sigma_{\mathcal{P}} \cup \{root\}$ and all $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$, i.e. after non-deterministically chosing a position in the tree, the automaton enters a designated state q_0, which occurs only once in each run.
- $(q_\alpha^{var}, \varepsilon) \in \Delta(q_0, R_{ij}^k, \tau)$ for all $k \in \mathcal{D}(\mathcal{P})$ and all $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ if and only if $head(R_{ij}) = \alpha \mapsto (\ldots)$, while in the designated state q_0, the automaton starts tracking the variable α, which is allocated at that position.

This routing automaton has a run over t, which labels one position by q_0 and a distinct one by q_f if and only if two positions in t allocate the same location. Notice that $B_{\mathcal{P}}$ has always a trivial run that starts and ends in the same position – since each position $p \in dom(t)$ allocates a variable α, and $\langle q_i, \varepsilon \rangle, \ldots, \langle q_0, p \rangle, \langle q_\alpha^{var}, p \rangle, \langle q_f, p \rangle$ is a valid run of $B_{\mathcal{P}}$. The predicate system has no double allocation if and only if these are the only possible runs of $B_{\mathcal{P}}$.

The existence of a run of $B_{\mathcal{P}}$ is captured by an MSO formula $\Phi_{B_{\mathcal{P}}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}})$, where r maps to the root of the unfolding tree, $\overrightarrow{\mathbf{X}}$ is the sequence of second order variables X_{ij}^k defined previously, T maps to the domain of the tree, and $\overrightarrow{\mathbf{Y}}$ is the sequence of

second-order variables Y_q, taken in some order, each of which maps to the set of tree positions visited by the automaton while in state $q \in Q'_{\mathcal{P}}$ – we denote by Y_0 and Y_f the variables from $\overrightarrow{\mathbf{Y}}$ that correspond to the states q_0 and q_f, respectively. Finally, we define the constraint: $no\_double\_alloc(r, \overrightarrow{\mathbf{X}}, T) \equiv \forall \overrightarrow{\mathbf{Y}} \ . \ \Phi_{B_{\mathcal{P}}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}}) \rightarrow Y_0 = Y_f$

4.5 Handling Parameters

The last issue to be dealt with is the role of the actual parameters passed to a recursively defined predicate $P_i(x_{i,1}, \ldots, x_{i,k})$ of $\mathcal{P}$, in a top-level formula. Then, for each parameter $x_{i,j}$ of P_i and each unfolding tree $t \in \mathcal{T}_i(\mathcal{P})$, there exists a path $\varepsilon = p_1, \ldots, p_m \in dom(t)$ and variables $\alpha_1, \ldots, \alpha_m \in LVar_{sl}$ such that $x_{i,j} \equiv \alpha_1$ and $\alpha_\ell^{p_\ell} = \alpha_{\ell+1}^{p_{\ell+1}}$ is a consequence of ϕ_t, for all $\ell = 1, \ldots, m-1$. Subsequently, there are three (not necessarily disjoint) possibilities:

1. $head(t(p_m)) \equiv \alpha_m \mapsto (\ldots)$, i.e. α_m is allocated
2. $head(t(p_m)) \equiv \beta \mapsto (\gamma_1, \ldots, \gamma_p, \ldots, \gamma_\ell)$, and $\alpha_m \equiv \gamma_p$, i.e. α_m is referenced
3. $\alpha_m \equiv x_{i,q}$ and $p_m = \varepsilon$, for some $1 \leq q \leq k$, i.e. α_m is another parameter $x_{i,q}$

Again, we use slightly modified routing automata (one for each of the case above) $C^{i,j}_{\mathcal{P},c} = (\Sigma_{\mathcal{P}}, Q''_{\mathcal{P}}, q_i, q_f, \Delta^{i,j}_c)$ for the cases $c = 1,2,3$, respectively. Here $Q''_{\mathcal{P}} = \{q^{var}_x \mid x \in LVar_{sl}\} \cup \{q^{sel}_s \mid s \in Sel\} \cup \{q^{i,a} \mid 1 \leq a \leq k\} \cup \{q_i, q_f\}$ and $\Delta^{i,j}_c$, $c = 1,2,3$ differ from the transitions of $A_{\mathcal{P}}$ in the following:

- $(q^{i,j}, \varepsilon) \in \Delta^{i,j}_x(q_i, root, ?)$, i.e. the automaton marks the root of the tree with a designated state $q^{i,j}$, that occurs only once on each run
- $(q^{var}_{x_{i,j}}, \varepsilon) \in \Delta^{i,j}_x(q^{i,j}, R^{-1}_{ik}, ?)$, for each rule R_{ik} of P_i, i.e. the automaton starts tracking the parameter variable $x_{i,j}$ beginning with the root of the tree
- $(q_f, \varepsilon) \in \Delta^{i,j}_1(q^{var}_\alpha, R^k_{ij}, \tau)$, for all $k \in \mathcal{D}(\mathcal{P})$, $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ iff $head(R_{ij}) \equiv \alpha \mapsto (\ldots)$ is the final rule for $C^{i,j}_{\mathcal{P},1}$
- $(q^{sel}_s, \varepsilon) \in \Delta^{i,j}_2(q^{var}_\gamma, R^k_{ij}, \tau)$, for all $k \in \mathcal{D}(\mathcal{P})$ and $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$ iff $head(R_{ij}) \equiv \alpha \mapsto (\beta_1, \ldots, \beta_s, \ldots, \beta_n)$ and $\gamma \equiv \beta_s$ i.e., q^{sel}_s is reached in the second case, when the tracked variable is referenced. After that, $C^{i,j}_{\mathcal{P},2}$ moves to the final state i.e., $(q_f, \varepsilon) \in \Delta^{i,j}_2(q^{sel}_s, \sigma, \tau)$ for all $s \in Sel$, all $\sigma \in \Sigma_{\mathcal{P}} \cup \{root\}$ and $\tau \in \Sigma_{\mathcal{P}} \cup \{?\}$
- $(q^{i,a}, \varepsilon) \in \Delta^{i,j}_3(q^{var}_{x_{i,a}}, root, ?)$ and $(q_f, \varepsilon) \in \Delta^{i,j}_3(q_{i,a}, root, ?)$, for each $1 \leq a \leq k$ and $a \neq j$ i.e., are the final moves for $C^{i,j}_{\mathcal{P},3}$

The outcome of this construction are MSO formulae $\Phi_{C^{i,j}_{\mathcal{P},c}}(r, \overrightarrow{\mathbf{X}}, T, \overrightarrow{\mathbf{Y}})$, for $c = 1,2,3$, where r maps to the root of the unfolding tree, respectively, $\overrightarrow{\mathbf{X}}$ is the sequence of second order variables X^k_{ij} defined previously, T maps to the domain of the tree, and $\overrightarrow{\mathbf{Y}}$ is the sequence of second order variables corresponding to states of $Q''_{\mathcal{P}}$ – we denote by $Y_f, Y^{i,a}, Y^{sel}_s \in \overrightarrow{\mathbf{Y}}$ the variables corresponding to the states q_f, $q^{i,a}$, and q^{sel}_s, respectively. The parameter $x_{i,j}$ of P_i is assigned by the following MSO constraints:

$$param^1_{i,j}(r,\overrightarrow{\mathbf{X}},T) \equiv \exists \overrightarrow{\mathbf{Y}} \;.\; \Phi_{C^{i,j}_{\mathcal{P},1}} \wedge Y_0^{i,j}(\overline{x}_{i,j}) \wedge \forall y \;.\; Y_f(y) \rightarrow \overline{x}_{i,j} = y$$
$$param^2_{i,j}(r,\overrightarrow{\mathbf{X}},T) \equiv \exists \overrightarrow{\mathbf{Y}} \;.\; \Phi_{C^{i,j}_{\mathcal{P},2}} \wedge Y_0^{i,j}(\overline{x}_{i,j}) \wedge \bigwedge_{s\in Sel} \forall y \;.\; Y_s^{sel}(y) \rightarrow edge_s(y,\overline{x}_{i,j})$$
$$param^3_{i,j}(r,\overrightarrow{\mathbf{X}},T) \equiv \exists \overrightarrow{\mathbf{Y}} \;.\; \Phi_{C^{i,j}_{\mathcal{P},3}} \wedge Y_0^{i,j}(\overline{x}_{i,j}) \wedge \bigwedge_{1\leq a\leq k} \forall y \;.\; Y^{i,a}(y) \rightarrow \overline{x}_{i,j} = \overline{x}_{i,a}$$

where $\overline{x}_{i,j}$ is the first-order MSO variable corresponding to the SL parameter $x_{i,j}$. Finally, the constraint $param_{i,j}$ is conjunction of the $param^c_{i,j}$, $c = 1,2,3$ formulae.

4.6 Translating Top Level SLRD$_{btw}$ Formulae to MSO

We define the MSO formula corresponding to a predicate $P_i(x_{i,1},\ldots,x_{i,n_i})$, of a system of recursive definitions $\mathcal{P} = \{P_1,\ldots,P_n\}$:

$$\overline{P_i}(\overline{x}_{i,1},\ldots,\overline{x}_{i,n_i},T) \equiv \exists r \exists \overrightarrow{\mathbf{X}} \;.\; backbone_i(r,\overrightarrow{\mathbf{X}},T) \wedge inner\_edges(r,\overrightarrow{\mathbf{X}},T) \wedge no\_double\_alloc(r,\overrightarrow{\mathbf{X}},T) \wedge \bigwedge_{1\leq j\leq n_i} param_{i,j}(r,\overrightarrow{\mathbf{X}},T)$$

The following lemma is needed to establish the correctness of our construction.

Lemma 5. *For any state $S = \langle s,h\rangle$, any interpretation $\iota : LVar_{sl} \rightarrow_{fin} Loc$, and any recursively defined predicate $P_i(x_1,\ldots,x_n)$, we have $S,\iota \models_{sl} P_i(x_1,\ldots,x_n)$ if and only if $S,\overline{\iota},\nu[T \leftarrow dom(h)] \models_{mso} \overline{P_i}(\overline{x_1},\ldots,\overline{x_k},T) \wedge Heap(T)$, where $\overline{\iota} : LVar_{mso} \rightharpoonup_{fin} Loc$ is an interpretation of first order variables, such that $\overline{\iota}(x_u) = s(u)$, for all $u \in PVar$, and $\overline{\iota}(\overline{x}) = \iota(x)$, for all $x \in LVar_{sl}$, and $\nu : LVar_{mso} \rightharpoonup_{fin} 2^{Loc}$ is any interpretation of second-order variables.*

Recall that a top level SLRD$_{btw}$ formula is of the form: $\varphi \equiv \exists \mathbf{z} \;.\; \phi(\mathbf{y}_0) * P_{i_1}(\mathbf{y}_1) * \ldots P_{i_k}(\mathbf{y}_k)$, where $1 \leq i_1,\ldots,i_k \leq n$, and $\mathbf{y}_j \subseteq \mathbf{z}$, for all $j = 0,1,\ldots,k$. We define the MSO formula:

$$\overline{\varphi}(X) \equiv \exists \overline{\mathbf{z}} \exists X_{0,\ldots,k} \;.\; \overline{\phi}(\overline{\mathbf{y}_0},X_0) \wedge \overline{P_{i_1}}(\overline{\mathbf{y}_1},X_1) \wedge \ldots \wedge \overline{P_{i_k}}(\overline{\mathbf{y}_k},X_k) \wedge \Pi(X_0,X_1,\ldots,X_k,X)$$

Theorem 3. *For any state S and any closed SLRD$_{btw}$ formula φ we have that $S \models_{sl} \varphi$ if and only if $S \models_{mso} \exists X \;.\; \overline{\varphi}(X) \wedge Heap(X)$.*

Theorem 2 and the above theorem prove decidability of satisfiability and entailment problems for SLRD$_{btw}$, by reduction to MSO over states of bounded tree width.

5 Conclusions and Future Work

We defined a fragment of Separation Logic with Recursive Definitions, capable of describing general unbounded mutable data structures, such as trees with parent pointers and linked leaves. The logic is shown to be decidable for satisfiability and entailment, by reduction to MSO over graphs of bounded tree width. We conjecture that the complexity of the decision problems for this logic is elementary, and plan to compute tight upper bounds, in the near future.

Acknowledgement. This work was supported by the Czech Science Foundation (project P103/10/0306) and French National Research Agency (project VERIDYC ANR-09-SEGI-016). We also acknowledge Tomáš Vojnar, Lukáš Holík and the anonymous reviewers for their valuable comments.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
3. Bojańczyk, M.: Tree-walking automata. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 1–2. Springer, Heidelberg (2008)
4. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 178–195. Springer, Heidelberg (2009)
5. Bozga, M., Iosif, R., Perarnau, S.: Quantitative separation logic and programs with lists. J. Autom. Reasoning 45(2), 131–156 (2010)
6. Brotherston, J., Kanovich, M.: Undecidability of propositional separation logic and its neighbours. In: Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, pp. 130–139 (2010)
7. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011)
8. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011)
9. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
10. Enea, C., Saveluc, V., Sighireanu, M.: Compositional invariant checking for overlaid and nested linked lists. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 129–148. Springer, Heidelberg (2013)
11. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: The boundary between decidability and undecidability for transitive-closure logics. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 160–174. Springer, Heidelberg (2004)
12. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. CoRR abs/1301.5139 (2013)
13. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Proc. of POPL 2011. ACM (2011)
14. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Proc. of POPL 2011 (2011)
15. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proc. of PLDI 2001 (June 2001)
16. Nguyen, H.H., Chin, W.-N.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008)
17. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS 2002. IEEE CS Press (2002)
18. Seese, D.: The structure of models of decidable monadic theories of graphs. Annals of Pure and Applied Logic 53(2), 169–195 (1991)
19. Yorsh, G., Rabinovich, A., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 94–110. Springer, Heidelberg (2006)

Appendix E

Fully Automated Shape Analysis Based on Forest Automata

Fully Automated Shape Analysis Based on Forest Automata

Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Forest automata (FA) have recently been proposed as a tool for shape analysis of complex heap structures. FA encode sets of tree decompositions of heap graphs in the form of tuples of tree automata. In order to allow for representing complex heap graphs, the notion of FA allowed one to provide user-defined FA (called boxes) that encode repetitive graph patterns of shape graphs to be used as alphabet symbols of other, higher-level FA. In this paper, we propose a novel technique of automatically learning the FA to be used as boxes that avoids the need of providing them manually. Further, we propose a significant improvement of the automata abstraction used in the analysis. The result is an efficient, fully-automated analysis that can handle even as complex data structures as skip lists, with the performance comparable to state-of-the-art fully-automated tools based on separation logic, which, however, specialise in dealing with linked lists only.

1 Introduction

Dealing with programs that use complex dynamic linked data structures belongs to the most challenging tasks in formal program analysis. The reason is a necessity of coping with infinite sets of reachable heap configurations that have a form of complex graphs. Representing and manipulating such sets in a sufficiently general, efficient, and automated way is a notoriously difficult problem.

In [6], a notion of *forest automata* (FA) has been proposed for representing sets of reachable configurations of programs with complex dynamic linked data structures. FA have a form of tuples of *tree automata* (TA) that encode sets of heap graphs decomposed into tuples of *tree components* whose leaves may refer back to the roots of the components. In order to allow for dealing with complex heap graphs, FA may be *hierarchically nested* by using them as alphabet symbols of other, higher-level FA. Alongside the notion of FA, a shape analysis applying FA in the framework of *abstract regular tree model checking* (ARTMC) [2] has been proposed in [6] and implemented in the Forester tool. ARTMC accelerates the computation of sets of reachable program configurations represented by FA by abstracting their component TA, which is done by collapsing some of their states. The analysis was experimentally shown to be capable of proving memory safety of quite rich classes of heap structures as well as to be quite efficient. However, it relied on the user to provide the needed nested FA—called *boxes*—to be used as alphabet symbols of the top-level FA.

In this paper, we propose a new shape analysis based on FA that avoids the need of manually providing the appropriate boxes. For that purpose, we propose a technique of automatically *learning* the FA to be used as boxes. The basic principle of the learning

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 740–755, 2013.

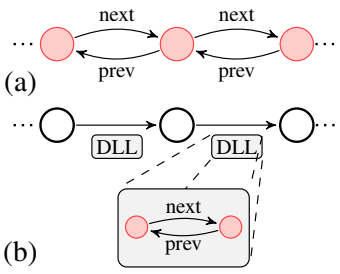


Fig. 1. (a) A DLL, (b) a hierarchical encoding of a DLL

stems from the reason for which boxes were originally introduced into FA. In particular, FA must have a separate component TA for each node (called a *join*) of the represented graphs that has multiple incoming edges. If the number of joins is unbounded (as, e.g., in doubly linked lists, abbreviated as DLLs below), unboundedly many component TA are needed in flat FA. However, when some of the edges are hidden in a box (as, e.g., the prev and next links of DLLs in Fig. 1) and replaced by a single box-labelled edge, a finite number of component TA may suffice. Hence, the basic idea of our learning is to identify subgraphs of the FA-represented graphs that contain at least one join, and when they are enclosed—or, as we say later on, *folded*—into a box, the in-degree of the join decreases.

There are, of course, many ways to select the above mentioned subgraphs to be used as boxes. To choose among them, we propose several criteria that we found useful in a number of experiments. Most importantly, the boxes must be *reusable* in order to allow eliminating as many joins as possible. The general strategy here is to choose boxes that are *simple* and *small* since these are more likely to correspond to graph patterns that appear repeatedly in typical data structures. For instance, in the already mentioned case of DLLs, it is enough to use a box enclosing a single pair of next/prev links. On the other hand, as also discussed below, too simple boxes are sometimes not useful either.

Further, we propose a way how box learning can be efficiently integrated into the main analysis loop. In particular, we do not use the perhaps obvious approach of incrementally building a *database of boxes* whose instances would be sought in the generated FA. We found this approach inefficient due to the costly operation of finding instances of different boxes in FA-represented graphs. Instead, we always try to identify which subgraphs of the graphs represented by a given FA could be folded into a box, followed by looking into the so-far built database of boxes whether such a box has already been introduced or not. Moreover, this approach has the advantage that it allows one to use simple language inclusion checks for *approximate box folding*, replacing a set of subgraphs that appear in the graphs represented by a given FA by a larger set, which sometimes greatly accelerates the computation. Finally, to further improve the efficiency, we interleave the process of box learning with the *automata abstraction* into a single iterative process. In addition, we propose an FA-specific improvement of the basic automata abstraction which *accelerates the abstraction* of an FA using components of other FA. Intuitively, it lets the abstraction synthesize an invariant faster by allowing it to combine information coming from different branches of the symbolic computation.

We have prototyped the proposed techniques in Forester and evaluated it on a number of challenging case studies. The results show that the obtained approach is both quite general as well as efficient. We were, e.g., able to fully-automatically analyse programs with 2-level and 3-level skip lists, which, according to the best of our knowledge, no other fully-automated analyser can handle. On the other hand, our implementation achieves performance comparable and sometimes even better than that of Predator [4]

(a winner of the heap manipulation division of SV-COMP'13) on list manipulating programs despite being able to handle much more general classes of heap graphs.

Related Work. As discussed already above, we propose a new shape analysis based upon the notion of forest automata introduced in [6]. The new analysis is extended by a mechanism for automatically learning the needed nested FA, which is carefully integrated into the main analysis loop in order to maximize its efficiency. Moreover, we formalize the abstraction used in [6], which was not done in [6], and subsequently significantly refine it in order to improve both its generality as well as efficiency.

From the point of view of efficiency and degree of automation, the main alternative to our approach is the fully-automated use of separation logic with inductive list predicates as implemented in Space Invader [12] or SLAyer [1]. These approaches are, however, much less general than our approach since they are restricted to programs over certain classes of linked lists (and cannot handle even structures such as linked lists with data pointers pointing either inside the list nodes or optionally outside of them, which we can easily handle as discussed later on). A similar comparison applies to the Predator tool inspired by separation logic but using purely graph-based algorithms [4]. The work [9] on overlaid data structures mentions an extension of Space Invader to trees, but this extension is of a limited generality and requires some manual help.

In [5], an approach for synthesising inductive predicates in separation logic is proposed. This approach is shown to handle even tree-like structures with additional pointers. One of these structures, namely, the so-called mcf trees implementing trees whose nodes have an arbitrary number of successors linked in a DLL, is even more general than what can in principle be described by hierarchically nested FA (to describe mcf trees, recursively nested FA or FA based on hedge automata would be needed). On the other hand, the approach of [5] seems quite dependent on exploiting the fact that the encountered 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), which is close to providing an inductive definition of the data structure.

The work [10] proposes an approach which uses separation logic for generating numerical abstractions of heap manipulating programs allowing for checking both their safety as well as termination. The described experiments include even verification of programs with 2-level skip lists. However, the work still expects the user to manually provide an inductive definition of skip lists in advance. Likewise, the work [3] based on the so-called separating shape graphs reports on verification of programs with 2-level skip lists, but it also requires the user to come up with summary edges to be used for summarizing skip list segments, hence basically with an inductive definition of skip lists. Compared to [10,3], we did not have to provide any manual aid whatsoever to our technique when dealing with 2-level as well as 3-level skip lists in our experiments.

A concept of inferring graph grammar rules for the heap abstraction proposed in [8] has recently appeared in [11]. However, the proposed technique can so far only handle much less general structures than in our case.

2 Forest Automata

Given a word $\alpha = a_1 \dots a_n, n \geq 1$, we write α_i to denote its i-th symbol a_i. Given a total map $f : A \to B$, we use $dom(f)$ to denote its domain A and $img(f)$ to denote its image.

Graphs. A *ranked alphabet* is a finite set of symbols Σ associated with a mapping $\# : \Sigma \to \mathbb{N}_0$ that assigns ranks to symbols. A (directed, ordered, labelled) *graph* over Σ is a total map $g : V \to \Sigma \times V^*$ which assigns to every *node* $v \in V$ (1) a *label* from Σ, denoted as $\ell_g(v)$, and (2) a sequence of *successors* from V^*, denoted as $S_g(v)$, such that $\#\ell_g(v) = |S_g(v)|$. We drop the subscript g if no confusion may arise. Nodes v with $S(v) = \varepsilon$ are called *leaves*. For any $v \in V$ such that $g(v) = (a, v_1 \cdots v_n)$, we call the pair $v \mapsto (a, v_1 \cdots v_n)$ an *edge* of g. The *in-degree* of a node in V is the overall number of its occurrences in $g(v)$ across all $v \in V$. The nodes of a graph g with an in-degree larger than one are called *joins* of g.

A *path* from v to v' in g is a sequence $p = v_0, i_1, v_1, \ldots, i_n, v_n$ where $v_0 = v$, $v_n = v'$, and for each $j : 1 \leq j \leq n$, v_j is the i_j-th successor of v_{j-1}. The *length* of p is defined as $\mathit{length}(p) = n$. The *cost* of p is the sequence $i_1, \ldots, i_n$. We say that p is cheaper than another path p' iff the cost of p is lexicographically smaller than that of p'. A node u is *reachable* from a node v iff there is a path from v to u or $u = v$. A graph g is *accessible* from a node v iff all its nodes are reachable from v. The node v is then called the *root* of g. A *tree* is a graph t which is either empty, or it has exactly one root and each of its nodes is the i-th successor of at most one node v for some $i \in \mathbb{N}$.

Forests. Let $\Sigma \cap \mathbb{N} = \emptyset$. A Σ-labelled *forest* is a sequence of trees $t_1 \cdots t_n$ over $(\Sigma \cup \{1, \ldots, n\})$ where $\forall 1 \leq i \leq n : \#i = 0$. Leaves labelled by $i \in \mathbb{N}$ are called *root references*.

The forest $t_1 \cdots t_n$ represents the graph $\otimes t_1 \cdots t_n$ obtained by uniting the trees of $t_1 \cdots t_n$, assuming w.l.o.g. that their sets of nodes are disjoint, and interconnecting their roots with the corresponding root references. Formally, $\otimes t_1 \cdots t_n$ contains an edge $v \mapsto (a, v_1 \cdots v_m)$ iff there is an edge $v \mapsto (a, v'_1 \cdots v'_m)$ of some tree $t_i, 1 \leq i \leq n$, s.t. for all $1 \leq j \leq m$, $v_j = \mathit{root}(t_k)$ if v'_j is a root reference with $\ell(v'_j) = k$, and $v_j = v'_j$ otherwise.

Tree Automata. A (finite, non-deterministic, top-down) *tree automaton* (TA) is a quadruple $A = (Q, \Sigma, \Delta, R)$ where Q is a finite set of *states*, $R \subseteq Q$ is a set of *root states*, Σ is a ranked alphabet, and Δ is a set of *transition rules*. Each transition rule is a triple of the form $(q, a, q_1 \ldots q_n)$ where $n \geq 0$, $q, q_1, \ldots, q_n \in Q$, $a \in \Sigma$, and $\#a = n$. In the special case where $n = 0$, we speak about the so-called *leaf rules*.

A *run* of A over a tree t over Σ is a mapping $\rho : \mathit{dom}(t) \to Q$ s.t. for each node $v \in \mathit{dom}(t)$ where $q = \rho(v)$, if $q_i = \rho(S(v)_i)$ for $1 \leq i \leq |S(v)|$, then Δ has a rule $q \to \ell(v)(q_1 \ldots q_{|S(v)|})$. We write $t \Longrightarrow_\rho q$ to denote that ρ is a run of A over t s.t. $\rho(\mathit{root}(t)) = q$. We use $t \Longrightarrow q$ to denote that $t \Longrightarrow_\rho q$ for some run ρ. The *language* of a state q is defined by $L(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of A is defined by $L(A) = \bigcup_{q \in R} L(q)$.

Graphs and Forests with Ports. We will further work with graphs with designated input and output points. An *io-graph* is a pair (g, ϕ), abbreviated as g_ϕ, where g is a graph and $\phi \in \mathit{dom}(g)^+$ a sequence of *ports* in which ϕ_1 is the *input port* and $\phi_2 \cdots \phi_{|\phi|}$ is a sequence of *output ports* such that the occurrence of ports in ϕ is unique. Ports and joins of g are called *cut-points* of g_ϕ. We use $\mathit{cps}(g_\phi)$ to denote all cut-points of g_ϕ. We say that g_ϕ is *accessible* if it is accessible from the input port ϕ_1.

An io-forest is a pair $f = (t_1 \cdots t_n, \pi)$ s.t. $n \geq 1$ and $\pi \in \{1, \ldots, n\}^+$ is a sequence of port indices, π_1 is the *input index*, and $\pi_2 \ldots \pi_{|\pi|}$ is a sequence of *output indices*, with no repetitions of indices in π. An io-forest encodes the io-graph $\otimes f$ where the ports of $\otimes t_1 \cdots t_n$ are roots of the trees defined by π, i.e., $\otimes f = (\otimes t_1 \cdots t_n, \mathit{root}(t_{\pi_1}) \cdots \mathit{root}(t_{\pi_n}))$.

Forest Automata. A *forest automaton* (FA) over Σ is a pair $F = (A_1 \cdots A_n, \pi)$ where $n \geq 1$, $A_1 \cdots A_n$ is a sequence of tree automata over $\Sigma \cup \{1, \ldots, n\}$, and $\pi \in \{1, \ldots, n\}^+$ is a sequence of port indices as defined for io-forests. The *forest language* of F is the set of io-forests $L_f(F) = L(A_1) \times \cdots \times L(A_n) \times \{\pi\}$, and the *graph language* of F is the set of io-graphs $L(F) = \{\otimes f \mid f \in L_f(F)\}$.

Structured Labels. We will further work with alphabets where symbols, called *structured labels*, have an inner structure. Let Γ be a ranked alphabet of *sub-labels*, ordered by a total ordering $\sqsubset_\Gamma$. We will work with graphs over the alphabet 2^Γ where for every symbol $A \subseteq \Gamma$, $\#A = \sum_{a \in A} \#a$. Let $e = v \mapsto (\{a_1, \ldots, a_m\}, v_1 \cdots v_n)$ be an edge of a graph g where $n = \sum_{1 \leq i \leq m} \#a_i$ and $a_1 \sqsubset_\Gamma a_2 \sqsubset_\Gamma \cdots \sqsubset_\Gamma a_m$. The triple $e\langle i \rangle = v \to (a_i, v_k \cdots v_l)$, $1 \leq i \leq m$, from the sequence $e\langle 1 \rangle = v \to (a_1, v_1 \cdots v_{\#a_1}), \ldots, e\langle m \rangle = v \to (a_m, v_{n-\#a_m+1} \cdots v_n)$ is called the i-th *sub-edge* of e (or the i-th sub-edge of v in g). We use $SE(g)$ to denote the set of all sub-edges of g. We say that a node v of a graph is *isolated* if it does not appear within any sub-edge, neither as an origin (i.e., $\ell(v) = \emptyset$) nor as a target. A graph g without isolated nodes is unambiguously determined by $SE(g)$ and vice versa (due to the total ordering $\sqsubset_\Gamma$ and since g has no isolated nodes). We further restrict ourselves to graphs with structured labels and without isolated nodes.

A counterpart of the notion of sub-edges in the context of rules of TA is the notion of rule-terms, defined as follows: Given a rule $\delta = (q, \{a_1, \ldots, a_m\}, q_1 \cdots q_n)$ of a TA over structured labels of 2^Γ, *rule-terms* of δ are the terms $\delta\langle 1 \rangle = a_1(q_1 \cdots q_{\#a_1}), \ldots, \delta\langle m \rangle = a_m(q_{n-\#a_m+1} \cdots q_n)$ where $\delta\langle i \rangle, 1 \leq i \leq m$, is called the *i-th rule-term of* δ.

Forest Automata of a Higher Level. We let Γ_1 be the set of all forest automata over 2^Γ and call its elements forest automata over Γ *of level 1*. For $i > 1$, we define Γ_i as the set of all forest automata over ranked alphabets $2^{\Gamma \cup \Delta}$ where $\Delta \subseteq \Gamma_{i-1}$ is any nonempty finite set of FA of level $i-1$. We denote elements of Γ_i as forest automata over Γ *of level* i. The rank $\#F$ of an FA F in these alphabets is the number of its output port indices. When used in an FA F over $2^{\Gamma \cup \Delta}$, the forest automata from Δ are called *boxes* of F. We write Γ_* to denote $\cup_{i \geq 0} \Gamma_i$ and assume that Γ_* is ordered by some total ordering $\sqsubset_{\Gamma_*}$.

An FA F of a higher level over Γ accepts graphs where forest automata of lower levels appear as sub-labels. To define the semantics of F as a set of graphs over Γ, we need the following operation of *sub-edge replacement* where a sub-edge of a graph is substituted by another graph. Intuitively, the sub-edge is removed, and its origin and targets are identified with the input and output ports of the substituted graph, respectively.

Formally, let g be a graph with an edge $e \in g$ and its i-th sub-edge $e\langle i \rangle = v_1 \to (a, v_2 \cdots v_n), 1 \leq i \leq |S_g(v_1)|$. Let g'_ϕ be an io-graph with $|\phi| = n$. Assume w.l.o.g. that $dom(g) \cap dom(g') = \emptyset$. The sub-edge $e\langle i \rangle$ can be replaced by g' provided that $\forall 1 \leq j \leq n : \ell_g(v_j) \cap \ell_{g'}(\phi_j) = \emptyset$, which means that the node $v_j \in dom(g)$ and the corresponding port $\phi_j \in dom(g')$ do not have successors reachable over the same symbol. If the replacement can be done, the result, denoted $g[g'_\phi / e\langle i \rangle]$, is the graph g_n in the sequence $g_0, \ldots, g_n$ of graphs defined as follows: $SE(g_0) = SE(g) \cup SE(g') \setminus \{e\langle i \rangle\}$, and for each $j : 1 \leq j \leq n$, the graph g_j arises from g_{j-1} by (1) deriving a graph h by replacing the origin of the sub-edges of the j-th port ϕ_j of g' by v_j, (2) redirecting edges leading to ϕ_j to v_j, i.e., replacing all occurrences of ϕ_j in $img(h)$ by v_j, and (3) removing ϕ_j.

If the symbol a above is an FA and $g'_\phi \in L(a)$, we say that $h = g[g'_\phi / e\langle i \rangle]$ is an *unfolding* of g, written $g \prec h$. Conversely, we say that g arises from h by *folding* g'_ϕ into

$e\langle i\rangle$. Let $\prec^*$ be the reflexive transitive closure of $\prec$. The Γ-*semantics* of g is then the set of graphs g' over Γ s.t. $g \prec^* g'$, denoted $[\![g]\!]_\Gamma$, or just $[\![g]\!]$ if no confusion may arise. For an FA F of a higher level over Γ, we let $[\![F]\!] = \bigcup_{g_\phi \in L(F)} ([\![g]\!] \times \{\phi\})$.

Canonicity. We call an io-forest $f = (t_1 \cdots t_n, \pi)$ *minimal* iff the roots of the trees $t_1 \cdots t_n$ are the cut-points of $\otimes f$. A minimal forest representation of a graph is unique up to reordering of $t_1 \cdots t_n$. Let the *canonical ordering* of cut-points of $\otimes f$ be defined by the cost of the cheapest paths leading from the input port to them. We say that f is *canonical* iff it is minimal, $\otimes f$ is accessible, and the trees within $t_1 \cdots t_n$ are ordered by the canonical ordering of their roots (which are cut-points of $\otimes f$). A canonical forest is thus a unique representation of an accessible io-graph. We say that an FA *respects canonicity* iff all forests from its forest language are canonical. Respecting canonicity makes it possible to efficiently test FA language inclusion by testing TA language inclusion of the respective components of two FA. This method is precise for FA of level 1 and sound (not always complete) for FA of a higher level [6].

In practice, we keep automata in the so called *state uniform* form, which simplifies maintaining of the canonicity respecting form [6] (and it is also useful when abstracting and "folding", as discussed in the following). It is defined as follows. Given a node v of a tree t in an io-forest, we define its *span* as the pair (α, V) where $\alpha \in \mathbb{N}^*$ is the sequence of labels of root references reachable from the root of t ordered according to the prices of the cheapest paths to them, and $V \subseteq \mathbb{N}$ is the set of labels of references which occur more than once in t. The state uniform form then requires that all nodes of forests from $L(F)$ that are labelled by the same state q in some accepting run of F have the same span, which we denote by $span(q)$.

3 FA-Based Shape Analysis

We now provide a high-level overview of the main loop of our shape analysis. The analysis automatically discovers memory safety errors (such as invalid dereferences of `null` or undefined pointers, double frees, or memory leaks) and provides an FA-represented over-approximation of the sets of heap configurations reachable at each program line. We consider sequential non-recursive C programs manipulating the heap. Each heap cell may have several *pointer selectors* and *data selectors* from some finite data domain (below, *PSel* denotes the set of pointer selectors, *DSel* denotes the set of data selectors, and $\mathbb{D}$ denotes the data domain).

Heap Representation. A single heap configuration is encoded as an io-graph $g_{\mathtt{sf}}$ over the ranked alphabet of structured labels 2^Γ with sub-labels from the ranked alphabet $\Gamma = PSel \cup (DSel \times \mathbb{D})$ with the ranking function that assigns each pointer selector 1 and each data selector 0. In this graph, an allocated memory cell is represented by a node v, and its internal structure of selectors is given by a label $\ell_g(v) \in 2^\Gamma$. Values of data selectors are stored directly in the structured label of a node as sub-labels from $DSel \times \mathbb{D}$, so, e.g., a singly linked list cell with the data value 42 and the successor node x_{next} may be represented by a node x such that $\ell_g(x) = \{\mathtt{next}(x_{next}), (\mathtt{data}, 42)(\varepsilon))\}$. Selectors with undefined values are represented such that the corresponding sub-labels are not in $\ell_g(x)$. The null value is modelled as the special node `null` such that $\ell_g(\mathtt{null}) = \emptyset$.

The input port sf represents a special node that contains the *stack frame* of the analysed function, i.e. a structure where selectors correspond to variables of the function.

In order to represent (infinite) *sets* of heap configurations, we use state uniform FA of a higher level to represent sets of canonical io-forests representing the heap configurations. The FA used as boxes are learnt during the analysis using the learning algorithm presented in Sec. 4.

Symbolic Execution. The verification procedure performs standard abstract interpretation with the abstract domain consisting of sets of state uniform FA (a single FA does not suffice as FA are not closed under union) representing sets of heap configurations at particular program locations. The computation starts from the initial heap configuration given by an FA for the io-graph $g_{\mathtt{sf}}$ where g comprises two nodes: null and sf where $\ell_g(\mathtt{sf}) = \emptyset$. The computation then executes abstract transformers corresponding to program statements until the sets of FA held at program locations stabilise. We note that abstract transformers corresponding to pointer manipulating statements are exact. Executing the abstract transformer $\tau_{\mathtt{op}}$ over a set of FA $\mathcal{S}$ is performed separately for every $F \in \mathcal{S}$. Some of boxes are first *unfolded* to uncover the accessed part of the heaps, then the update is performed. The detailed description of these steps can be found in [7].

At junctions of program paths, the analysis computes unions of sets of FA. At loop points, the union is followed by widening. The widening is performed by applying box *folding* and *abstraction* repeatedly in a loop on each FA from $\mathcal{S}$ until the result stabilises. An elaboration of these two operations, described in detail in Sec. 4 and 5 respectively, belongs to the main contribution of the presented paper.

4 Learning of Boxes

Sets of graphs with an unbounded number of joins can only be described by FA with the help of boxes. In particular, boxes allow one to replace (multiple) incoming sub-edges of a join by a single sub-edge, and hence lower the in-degree of the join. Decreasing the in-degree to 1 turns the join into an ordinary node. When a box is then used in a cycle of an FA, it effectively generates an unbounded number of joins.

The boxes are introduced by the operation of *folding* of an FA F which transforms F into an FA F' and a box B used in F' such that $[\![F]\!] = [\![F']\!]$. However, the graphs in $L(F')$ may contain less joins since some of them are hidden in the box B, which encodes a set of subgraphs containing a join and appearing repeatedly in the graphs of $L(F)$. Before we explain folding, we give a characterisation of subgraphs of graphs of $L(F)$ which we want to fold into a box B. Our choice of the subgraphs to be folded is a compromise between two high-level requirements. On the one hand, the folded subgraphs should contain incoming edges of joins and be as simple as possible in order to be reusable. On the other hand, the subgraphs should not be too small in order not to have to be subsequently folded within other boxes (in the worst case, leading to generation of unboundedly nested boxes). Ideally, the hierarchical structuring of boxes should respect the natural hierarchical structuring of the data structures being handled since if this is not the case, unboundedly many boxes may again be needed.

4.1 Knots of Graphs

A graph h is a *subgraph* of a graph g iff $SE(h) \subseteq SE(g)$. The *border* of h in g is the subset of the set $dom(h)$ of nodes of h that are incident with sub-edges in $SE(g) \setminus SE(h)$. A *trace* from a node u to a node v in a graph g is a set of sub-edges $t = \{e_0, \ldots, e_n\} \subseteq SE(g)$ such that $n \geq 1$, e_0 is an outgoing sub-edge of u, e_n is an incoming sub-edge of v, the origin of e_i is one of the targets of e_{i-1} for all $1 \leq i \leq n$, and no two sub-edges have the same origin. We call the origins of $e_1, \ldots, e_n$ the *inner nodes* of the trace. A trace from u to v is *straight* iff none of its inner nodes is a cut-point. A *cycle* is a trace from a node v to v. A *confluence* of g_ϕ is either a cycle of g_ϕ or it is the union of two disjoint traces starting at a node u, called the *base*, and ending in the node v, called the *tip* (for a cycle, the base and the tip coincide).

Given an io-graph g_ϕ, the *signature* of a sub-graph h of g is the minimum subset $sig(h)$ of $cps(g_\phi)$ that (1) contains $cps(g_\phi) \cap dom(h)$ and (2) all nodes of h, except the nodes of $sig(h)$ themselves, are reachable by straight traces from $sig(h)$. Intuitively, $sig(h)$ contains all cut-points of h plus the closest cut-points to h which lie outside of h but which are needed so that all nodes of h are reachable from the signature. Consider the example of the graph g_u in Fig. 2 in which cut-points are represented by •. The signature of g_u is the set $\{u, v\}$. The signature of the highlighted subgraph h is also equal to $\{u, v\}$. Given a set $U \subseteq cps(g_\phi)$, a *confluence of* U is a confluence of g_ϕ with the signature within U. Intuitively, the confluence of a set of cut-points U is a confluence whose cut-points belong to U plus in case the base is not a cut-point, then the closest cut-point from which the base is reachable is also from U. Finally, the *closure* of U is the smallest subgraph h of g_ϕ that (1) contains all confluences of U and (2) for every inner node v of a straight trace of h, it contains all straight traces from v to leaves of g. The closure of the signature $\{u, v\}$ of the graph g_u in Fig. 2 is the highlighted subgraph h. Intuitively, Point 1 includes into the closure all nodes and sub-edges that appear on straight traces between nodes of U apart from those that do not lie on any confluence (such as node u in Fig. 2). Note that nodes x and y in Fig. 2, which are leaves of g_u, are not in the closure as they are not reachable from an inner node of any straight trace of h. The *closure of a subgraph* h of g_ϕ is the closure of its signature, and h is *closed* iff it equals its closure.

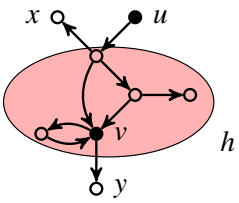


Fig. 2. Closure

Knots. For the rest of Sec. 4.1, let us fix an io-graph $g_\phi \in L(F)$. We now introduce the notion of a knot which summarises the desired properties of a subgraph k of g that is to be folded into a box. A *knot* k of g_ϕ is a subgraph of g such that: (1) k is a confluence, (2) k is the union of two knots with intersecting sets of sub-edges, or (3) k is the closure of a knot. A *decomposition* of a knot k is a set of knots such that the union of their sub-edges equals $SE(k)$. The *complexity of a decomposition* of k is the maximum of sizes of signatures of its elements. We define the *complexity of a knot* as the minimum of the complexities of its decompositions. A knot k of complexity n is an *optimal knot of complexity* n if it is maximal among knots of complexity n and if it has a root. The root must be reachable from the input port of g_ϕ by a trace that does not intersect with sub-edges of the optimal knot. Notice that the requirement of maximality implies that optimal knots are closed.

The following lemma, proven in [7], implies that optimal knots are uniquely identified by their signatures, which is crucial for the folding algorithm presented later.

Lemma 1. *The signature of an optimal knot of g_ϕ equals the signature of its closure.*

Next, we explain what is the motivation behind the notion of an optimal knot:

Confluences. As mentioned above, in order to allow one to eliminate a join, a knot must contain some join v together with at least one incoming sub-edge in case the knot is based on a loop and at least two sub-edges otherwise. Since g_ϕ is accessible (meaning that there do not exist any traces that cannot be extended to start from the same node), the edge must belong to some confluence c of g_ϕ. If the folding operation does not fold the entire c, then a new join is created on the border of the introduced box: one of its incoming sub-edges is labelled by the box that replaces the folded knot, another one is the last edge of one of the traces of c. Confluences are therefore the smallest subgraphs that can be folded in a meaningful way.

Uniting knots. If two different confluences c and c' share an edge, then after folding c, the resulting edge shares with c' two nodes (at least one being a target node), and thus c' contains a join of g_ϕ. To eliminate this join too, both confluences must be folded together. A similar reasoning may be repeated with knots in general. Usefulness of this rule may be illustrated by an example of the set of lists with head pointers. Without uniting, every list would generate a hierarchy of knots of the same depth as the length of the list, as illustrated in Fig. 3. This is clearly impractical since the entire set could not be represented using finitely many boxes. Rule 2 unites all knots into one that contains the entire list, and the set of all such knots can then be represented by a single FA (containing a loop accepting the inner nodes of the lists).

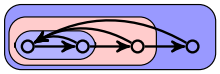


Fig. 3. A list with head pointers

Complexity of knots. The notion of complexity is introduced to limit the effect of Rule 2 of the definition of a knot, which unites knots that share a sub-edge, and to hopefully make it follow the natural hierarchical structuring of data structures. Consider, for instance, the case of singly-linked lists (SLLs) of cyclic doubly-linked lists (DLLs). In this case, it is natural to first fold the particular segments of the DLLs (denoted as DLSs below), i.e., to introduce a box for a single pair of next and prev pointers. This way, one effectively obtains SLLs of cyclic SLLs. Subsequently, one can fold the cyclic SLLs into a higher-level box. However, uniting all knots with a common sub-edge would create knots that contain entire cyclic DLLs (requiring unboundedly many joins inside the box). The reason is that in addition to the confluences corresponding to DLSs, there are confluences which traverse the entire cyclic DLLs and that share sub-edges with all DLSs (this is in particular the case of the two circular sequences consisting solely of next and prev pointers respectively). To avoid the undesirable folding, we exploit the notion of complexity and fold graphs in successive rounds. In each round we fold all optimal knots with the smallest complexity (as described in Sec. 4.2), which should correspond to the currently most nested, not yet folded, sub-structures. In the previous example, the algorithm starts by folding DLSs of complexity 2, because the complexity of the confluences in cyclic DLLs is given by the number of the DLSs they traverse.

Closure of knots. The closure is introduced for practical reasons. It allows one to identify optimal knots by their signatures, which is then used to simplify automata constructions that implement folding on the level of FA (cf. Sec. 4.2).

Root of an optimal knot. The requirement for an optimal knot k to have a root is to guarantee that if an io-graph h_ψ containing a box B representing k is accessible, then the io-graph $h_\psi[k/B]$ emerging by substituting k for a sub-edge labelled with B is accessible, and vice versa. It is also a necessary condition for the existence of a canonical forest representation of the knot itself (since one needs to order the cut-points w.r.t. the prices of the paths leading to them from the input port of the knot).

4.2 Folding in the Abstraction Loop

In this section, we describe the operation of folding together with the main abstraction loop of which folding is an integral part. The pseudo-code of the main abstraction loop is shown in Alg. 1. The algorithm modifies a set of FA until it reaches a fixpoint. Folding on line 5 is a sub-procedure of the algorithm which looks for substructures of FA that accept optimal knots, and replaces these substructures by boxes that represent the corresponding optimal knots. The operation of folding is itself composed of four consecutive steps: *Identifying indices*, *Splitting*, *Constructing boxes*, and *Applying boxes*. For space reasons, we give only an overview of the steps of the main abstraction loop and folding. Details may be found in [7].

```
1 Unfold solitaire boxes
2 repeat
3     Normalise
4     Abstract
5     Fold
6 until fixpoint
```

Alg. 1: Abstraction Loop

Unfolding of Solitaire Boxes. Folding is in practice applied on FA that accept partially folded graphs (only some of the optimal knots are folded). This may lead the algorithm to hierarchically fold data structures that are not hierarchical, causing the symbolic execution not to terminate. For example, consider a program that creates a DLL of an arbitrary length. Whenever a new DLS is attached, the folding algorithm would enclose it into a box together with the tail which was folded previously. This would lead to creation of a hierarchical structure of an unbounded depth (see Fig. 4), which would cause the symbolic execution to never reach a fixpoint. Intuitively, this is a situation when a repetition of subgraphs may be expressed by an automaton loop that iterates a box, but it is instead misinterpreted as a recursive nesting of graphs. This situation may happen when a newly created box contains another box that cannot be iterated since it does not appear on a loop (e.g, in Fig. 4 there is always one occurrence of a box encoding a shorter DLL fragment inside a higher-level box). This issue is addressed in the presented algorithm by first unfolding all occurrences of boxes that are not iterated by automata loops before folding is started.

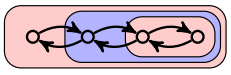


Fig. 4. DLL

Normalising. We define the *index* of a cut-point $u \in cps(g_\phi)$ as its position in the canonical ordering of cut-points of g_ϕ, and the *index* of a closed subgraph h of g_ϕ as the set of indices of the cut-points in $sig(h)$. The folding algorithm expects the input FA F to satisfy the property that all io-graphs of $L(F)$ have the same indices of closed knots. The reason is that folding starts by identifying the index of an optimal knot of an arbitrary io-graph from $L(F)$, and then it creates a box which accepts all closed subgraphs of the io-graphs from g_ϕ with the same index. We need a guarantee that *all* these subgraphs are indeed optimal knots. This guarantee can be achieved if the io-graphs from $L(F)$ have equivalent interconnections of cut-points, as defined below.

We define the relation $\sim_{g_\phi} \subseteq \mathbb{N} \times \mathbb{N}$ between indices of closed knots of g_ϕ such that $N \sim_{g_\phi} N'$ iff there is a closed knot k of g_ϕ with the index N and a closed knot k' with the index N' such that k and k' have intersecting sets of sub-edges. We say that two io-graphs g_ϕ and h_ψ are *interconnection equivalent* iff $\sim_{g_\phi} = \sim_{h_\psi}$.

Lemma 2. *Interconnection equivalent io-graphs have the same indices of optimal knots.*

Interconnection equivalence of all io-graphs in the language of an FA F is achieved by transforming F to the *interconnection respecting form*. This form requires that the language of every TA of the FA consists of interconnection equivalent trees (when viewing root references and roots as cut-points with corresponding indices). The transformation is described in [7]. The normalisation step also includes a transformation into the state uniform and canonicity respecting form.

Abstraction. We use abstraction described in Sec. 5 that preserves the canonicity respecting form of TA as well as their state uniformity. It may break interconnection uniformity, in which case it is followed by another round of normalisation. Abstraction is included into each round of folding for the reason that it leads to learning more general boxes. For instance, an FA encoding a cyclic list of one particular length is first abstracted into an FA encoding a set of cyclic lists of all lengths, and the entire set is then folded into a single box.

Identifying Indices. For every FA F entering this sub-procedure, we pick an arbitrary io-graph $g_\phi \in L(F)$, find all its optimal knots of the smallest possible complexity n, and extract their indices. By Lemma 2 and since F is normalised, indices of the optimal knots are the same for all io-graphs in $L(F)$. For every found index, the following steps fold all optimal knots with that index at once. Optimal knots of complexity n do not share sub-edges, the order in which they are folded is therefore not important.

Splitting. For an FA $F = (A_1 \cdots A_n, \pi)$ and an index I of an optimal knot found in the previous step, splitting transforms F into a (set of) new FA with the same language. The nodes of the borders of I-indexed optimal knots of io-graphs from $L(F)$ become roots of trees of io-forests accepted by the new FA. Let $s \in I$ be a position in F such that the s-indexed cut-points of io-graphs from $L(F)$ reach all the other I-indexed cut-points. The index s exists since an optimal knot has a root. Due to the definition of the closure, the border contains all I-indexed cut-points, with the possible exception of s. The s-th cut-point may be replaced in the border of the I-indexed optimal knot by the base e of the I-indexed confluence that is the first one reached from the s-th cut-point by a straight path. We call e the *entry*. The entry e is a root of the optimal knot, and the s-th cut-point is the only I-indexed cut-point that might be outside the knot. If e is indeed different from the s-th cut-point, then the s-th tree of forests accepted by F must be split into two trees in the new FA: The subtree rooted at the entry is replaced by a reference to a new tree. The new tree then equals the subtree of the original s-th tree rooted at the entry.

The construction is carried out as follows. We find all states and all of their rules that accept entry nodes. We denote such states and rules as entry states and rules. For every entry state q, we create a new FA F_q^0 which is a copy of F but with the s-th TA A_s split to a new s-th TA A'_s and a new $(n+1)$-th TA A_{n+1}. The TA A'_s is obtained from A_s by changing the entry rules of q to accept just a reference to the new $(n+1)$-th root and by removing entry rules of all other entry states (the entry states are processed separately in

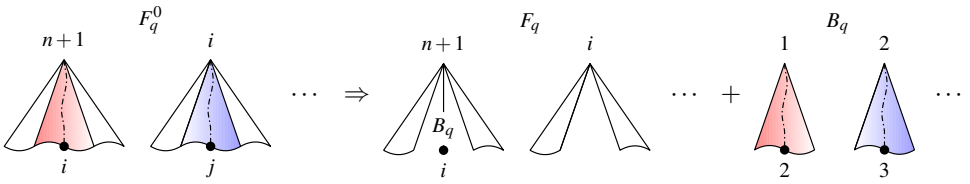


Fig. 5. Creation of F_q and B_q from F_q^0. The subtrees that contain references $i, j \in J$ are taken into B_q, and replaced by the B_q-labelled sub-edge in F_q.

order to preserve possibly different contexts of entry nodes accepted at different states). The new TA A_{n+1} is a copy of A_s but with the only accepting state being q. Note that the construction is justified since due to state uniformity, each node that is accepted by an entry rule and that does not appear below a node that is also accepted by an entry rule is an entry node. In the result, the set $J = (I \setminus \{s\}) \cup \{n+1\}$ contains the positions of the trees of forests of F_q^0 rooted at the nodes of the borders of I-indexed optimal knots.

Constructing Boxes. For every F_q^0 and J being the result of splitting F according to an index I, a box B_q is constructed from F_q^0. We transform TA of F_q^0 indexed by the elements of J. The resulting TA will accept the original trees up to that the roots are stripped from the children that cannot reach a reference to J. To turn these TA into an FA accepting optimal knots with the index I, it remains to order the obtained TA and define port indices, which is described in detail in [7]. Roughly, the input index of the box will be the position j to which we place the modified $(n+1)$-th TA of F_q^0 (the one that accepts trees rooted at the entry). The output indices are the positions of the TA with indices $J \setminus \{j\}$ in F_q^0 which accept trees rooted at cut-points of the border of the optimal knots.

Applying Boxes. This is the last step of folding. For every F_q^0, J, and B_q which are the result of splitting F according to an index I, we construct an FA F_q that accepts graphs of F where knots enclosed in B_q are substituted by a sub-edge with the label B_q. It is created from F_q^0 by (1) leaving out the parts of root rules of its TA that were taken into B_q, and (2) adding the rule-term $B_q(r_1, \ldots, r_m)$ to the rule-terms of root rules of the $(n+1)$-th component of F_q^0 (these are rules used to accept the roots of the optimal knots enclosed in B_q). The states $r_1, \ldots, r_m$ are fresh states that accept root references to the appropriate elements of J (to connect the borders of knots of B_q correctly to the graphs of F_q—the details may be found in [7]). The FA F_q now accepts graphs where optimal knots of graphs of $L(F)$ with the signature I are hidden inside B_q. Creation of B_q and of its counterpart F_q from F_q^0 is illustrated in Fig. 5 where $i, j, \ldots \in J$.

During the analysis, the discovered boxes must be stored in a database and tested for equivalence with the newly discovered ones since the alphabets of FA would otherwise grow with every operation of folding *ad infinitum*. That is, every discovered box is given a unique name, and whenever a semantically equivalent box is folded, the newly created edge-term is labelled by that name. This step offers an opportunity for introducing another form of acceleration of the symbolic computation. Namely, when a box B is found by the procedure described above, and another box B' with a name N s.t. $[\![B']\!] \subset [\![B]\!]$ is already in the database, we associate the name N with B instead of with B' and restart the analysis (i.e., start the analysis from the scratch, remembering just the updated database of boxes). If, on the other hand, $[\![B]\!] \subseteq [\![B']\!]$, the folding is performed using the name N

of B', thus overapproximating the semantics of the folded FA. As presented in Sec. 6, this variant of the procedure, called *folding by inclusion*, performs in some difficult cases significantly better than the former variant, called *folding by equivalence*.

5 Abstraction

The abstraction we use in our analysis is based on the general techniques described in the framework of abstract regular (tree) model checking [2]. We, in particular, build on the *finite height abstraction* of TA. It is parameterised by a height $k \in \mathbb{N}$, and it collapses TA states q, q' iff they accept trees with the same sets of prefixes of the height at most k (the prefix of height k of a tree is a subgraph of the tree which contains all paths from the root of length at most k). This defines an equivalence on states denoted by $\approx_k$. The equivalence $\approx_k$ is further refined to deal with various features special for FA. Namely, it has to work over tuples of TA and cope with the interconnection of the TA via root references, with the hierarchical structuring, and with the fact that we use a *set* of FA instead of a single FA to represent the abstract context at a particular program location.

Refinements of $\approx_k$. First, in order to maintain the same basic shape of the heap after abstraction (such that no cut-point would, e.g., suddenly appear or disappear), we refine $\approx_k$ by requiring that equivalent states must have the same spans (as defined in Sec. 2). When applied on $\approx_1$, which corresponds to equivalence of data types, this refinement provided enough precision for most of the case studies presented later on, with the exception of the most difficult ones, namely programs with skip lists [13]. To verify these programs, we needed to further refine the abstraction to distinguish automata states whenever trees from their languages encode tree components containing a different number of unique paths to some root reference, but some of these paths are hidden inside boxes. In particular, two states q, q' can be equivalent only if for every io-graph g_ϕ from the graph language of the FA, for every two nodes $u, v \in dom(g_\phi)$ accepted by q and q', respectively, in an accepting run of the corresponding TA, the following holds: For every $w \in cps(g_\phi)$, both u and v have the same number of outgoing sub-edges (selectors) in $[\![g_\phi]\!]$ which start a trace in $[\![g_\phi]\!]$ leading to w. According to our experiments, this refinement does not cost almost any performance, and hence we use it by default.

Abstraction for Sets of FA. Our analysis works with sets of FA. We observed that abstracting individual FA from a set of FA in isolation is sometimes slow since in each of the FA, the abstraction widens some selector paths only, and it takes a while until an FA in which all possible selector paths are widened is obtained. For instance, when analysing a program that creates binary trees, before reaching a fixpoint, the symbolic analysis generates many FA, each of them accepting a subset of binary trees with some of the branches restricted to a bounded length (e.g., trees with no right branches, trees with a single right branch of length 1, length 2, etc.). In such cases, it helps when the abstraction has an opportunity to combine information from several FA. For instance, consider an FA that encodes binary trees degenerated to an arbitrarily long left branch, and another FA that encodes trees degenerated to right branches only. Abstracting these FA in isolation has no effect. However, if the abstraction is allowed to collapse states from both of these FA, it can generate an FA accepting all possible branches.

Unfortunately, the natural solution to achieve the above, which is to unite FA before abstraction, cannot be used since FA are not closed under union (uniting TA component-

wise overapproximates the union). However, it is possible to enrich the automata structure of an FA F by TA states and rules of another one without changing the language of F, and in this way allow the abstraction to combine the information from both FA. In particular, before abstracting an FA $F = (A_1 \cdots A_n, \pi)$ from a set S of FA, we pre-process it as follows. (1) We pick automata $F' = (A'_1 \cdots A'_n, \pi) \in S$ which are compatible with F in that they have the same number of TA, the same port references, and for each $1 \leq i \leq n$, the root states of A'_i have the same spans as the root states of A_i. (2) For all such F' and each $1 \leq i \leq n$, we add rules and states of A'_i to A_i, but we keep the original set of root states of A_i. Since we assume that the sets of state of TAs of different FA are disjoint, the language of A_i stays the same, but its structure is enriched, which helps the abstraction to perform a coarser widening.

6 Experimental Results

We have implemented the above proposed techniques in the Forester tool and tested their generality and efficiency on a number of case studies. In the experiments, we compare two configurations of Forester, and we also compare the results of Forester with those of Predator [4], which uses a graph-based memory representation inspired by separation logic with higher-order list predicates. We do not provide a comparison with Space Invader [12] and SLAyer [1], based also on separation logic with higher-order list predicates, since in our experiments they were outperformed by Predator.

In the experiments, we considered programs with various types of lists (singly and doubly linked, cyclic, nested, with skip pointers), trees, and their combinations. In the case of skip lists, we had to slightly modify the algorithms since their original versions use an ordering on the data stored in the nodes of the lists (which we currently do not support) in order to guarantee that the search window delimited on some level of skip pointers is not left on any lower level of the skip pointers. In our modification, we added an additional explicit end-of-window pointer. We checked the programs for memory safety only, i.e., we did not check data-dependent properties.

Table 1 gives running times in seconds (the average of 10 executions) of the tools on our case studies. “Basic” stands for Forester with the abstraction applied on individual FA only and “SFA” stands for Forester with the abstraction for sets of FA. The value T means that the running time of the tool exceeded 30 minutes, and the value Err means that the tool reported a spurious error. The names of the examples in the table contain the name of the data structure manipulated in the program, which is “SLL” for singly linked lists, “DLL” for doubly linked lists (the “C” prefix denotes cyclic lists), “tree” for binary trees, “tree+parents” for trees with parent pointers. Nested variants of SLL (DLL) are named as “SLL (DLL) of” and the type of the nested structure. In particular, “SLL of 0/1 SLLs” stands for SLL of a nested SLL of length 0 or 1, and “SLL of 2CDLLs” stands for SLL whose each node is a root of two CDLLs. The “+head” flag stands for a list where each element points to the head of the list and the subscript “Linux” denotes the implementation of lists used in the Linux kernel, which uses type casts and a restricted pointer arithmetic. The “DLL+subdata” stands for a kind of a DLL with data pointers pointing either inside the list nodes or optionally outside of them. For a “skip list”, the subscript denotes the number of skip pointers. In the example “tree+stack”, a

Table 1. Results of the experiments

Example	basic	SFA	boxes	Predator	Example	basic	SFA	boxes	Predator
SLL (delete)	0.03	0.04		0.04	DLL (reverse)	0.04	0.06	1 / 1	0.03
SLL (bubblesort)	0.04	0.04		0.03	DLL (insert)	0.06	0.07	1 / 1	0.05
SLL (mergesort)	0.08	0.15		0.10	DLL (insertsort1)	0.35	0.40	1 / 1	0.11
SLL (insertsort)	0.05	0.05		0.04	DLL (insertsort2)	0.11	0.12	1 / 1	0.05
SLL (reverse)	0.03	0.03		0.03	DLL of CDLLs	5.67	1.25	8 / 7	0.22
SLL+head	0.05	0.05		0.03	DLL+subdata	0.06	0.09	- / 2	T
SLL of 0/1 SLLs	0.03	0.03		0.11	CDLL	0.03	0.03	1 / 1	0.03
$\text{SLL}_{\text{Linux}}$	0.03	0.03		0.03	tree	0.14	0.14		Err
SLL of CSLLs	2.07	0.73	3 / 4	0.12	tree+parents	0.18	0.21	2 / 2	T
SLL of $\text{2CDLLs}_{\text{Linux}}$	0.16	0.17	13 / 5	0.25	tree+stack	0.09	0.08		Err
skip list_2	0.66	0.42	- / 3	T	tree (DSW)	1.74	0.40		Err
skip list_3	T	9.14	- / 7	T	tree of CSLLs	0.32	0.42	- / 4	Err

randomly constructed tree is deleted using a stack, and "DSW" stands for the Deutsch-Schorr-Waite tree traversal (the Lindstrom variant). All experiments start with a random creation and end with a disposal of the specified structure while the indicated procedure (if any) is performed in between. The experiments were run on a machine with the Intel i7-2600 (3.40 GHz) CPU and 16 GiB of RAM.

The table further contains the column "boxes" where the value "X/Y" means that X manually created boxes were provided to the analysis that did not use learning while Y boxes were learnt when the box learning procedure was enabled. The value "-" of X means that we did not run the given example with manually constructed boxes since their construction was too tedious. If user-defined boxes are given to Forester in advance, the speedup is in most cases negligible, with the exception of "DLL of CDLLs" and "SLL of CSLLs", where it is up to 7 times. In a majority of cases, the learnt boxes were the same as the ones created manually. However, in some cases, such as "SLL of $\text{2CDLLs}_{\text{Linux}}$", the learning algorithm found a smaller set of more elaborate boxes than those provided manually.

In the experiments, we use folding by inclusion as defined in Sec. 4.2. For simpler cases, the performance matched the performance of folding by equivalence, but for the more difficult examples it was considerably faster (such as for "skip list_2" when the time decreased from 3.82 s to 0.66 s), and only when it was used the analysis of "skip list_3" succeeded. Further, the implementation folds optimal knots of the complexity ≤ 2 which is enough for the considered examples. Finally, note that the performance of Forester in the considered experiments is indeed comparable with that of Predator even though Forester can handle much more general data structures.

7 Conclusion

We have proposed a new shape analysis using forest automata which—unlike the previously known approach based on FA—is fully automated. For that purpose, we have proposed a technique of automatically learning FA called boxes to be used as alphabet symbols in higher-level FA when describing sets of complex heap graphs. We have also proposed a way how to efficiently integrate the learning with the main analysis

algorithm. Finally, we have proposed a significant improvement—both in terms of generality as well as efficiency—of the abstraction used in the framework. An implementation of the approach in the Forester tool allowed us to fully-automatically handle programs over quite complex heap structures, including 2-level and 3-level skip lists, which—to the best of our knowledge—no other fully-automated verification tool can handle. At the same time, the efficiency of the analysis is comparable with other state-of-the-art analysers even though they handle less general classes of heap structures.

For the future, there are many possible ways how the presented approach can be further extended. First, one can think of using recursive boxes or forest automata using hedge automata as their components in order to handle even more complex data structures (such as mcf trees). Another interesting direction is that of integrating FA-based heap analysis with some analyses for dealing with infinite non-pointer data domains (e.g., integers) or parallelism.

Acknowledgement. This work was supported by the Czech Science Foundation (projects P103/10/0306, 13-37876P), the Czech Ministry of Education, Youth, and Sports (project MSM 0021630528), the BUT FIT project FIT-S-12-1, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

References

1. Berdine, J., Cook, B., Ishtiaq, S.: Memory Safety for Systems-level Code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)
2. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular (Tree) Model Checking. STTT 14(2) (2012)
3. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape Analysis with Structural Invariant Checkers. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)
4. Dudka, K., Peringer, P., Vojnar, T.: Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
5. Guo, B., Vachharajani, N., August, D.I.: Shape Analysis with Inductive Recursion Synthesis. In: Proc. of PLDI 2007. ACM Press (2007)
6. 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)
7. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully Automated Shape Analysis Based on Forest Automata. Tech. rep. FIT-TR-2013-01, FIT BUT (2013)
8. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. ENTCS 266 (2010)
9. Lee, O., Yang, H., Petersen, R.: Program Analysis for Overlaid Data Structures. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 592–608. Springer, Heidelberg (2011)
10. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Automatic Numeric Abstractions for Heap-manipulating programs. In: Proc. of POPL 2010. ACM Press (2010)
11. Weinert, A.D.: Inferring Heap Abstraction Grammars. BSc thesis, RWTH Aachen (2012)
12. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
13. Pugh, W.: Skip Lists: A Probabilistic Alternative to Balanced Trees. Commun. ACM 33(6), 668–676 (1990)

Appendix F

Deciding Entailments in Inductive Separation Logic with Tree Automata

Deciding Entailments in Inductive Separation Logic with Tree Automata

Radu Iosif[1], Adam Rogalewicz[2], and Tomáš Vojnar[2]

[1] University Grenoble Alpes, CNRS, VERIMAG, Grenoble, France
[2] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Separation Logic (SL) with inductive definitions is a natural formalism for specifying complex recursive data structures, used in compositional verification of programs manipulating such structures. The key ingredient of any automated verification procedure based on SL is the decidability of the entailment problem. In this work, we reduce the entailment problem for a non-trivial subset of SL describing trees (and beyond) to the language inclusion of tree automata (TA). Our reduction provides tight complexity bounds for the problem and shows that entailment in our fragment is EXPTIME-complete. For practical purposes, we leverage from recent advances in automata theory, such as inclusion checking for non-deterministic TA avoiding explicit determinization. We implemented our method and present promising preliminary experimental results.

1 Introduction

Separation Logic (SL) [22] is a logical framework for describing recursive mutable data structures. The attractiveness of SL as a specification formalism comes from the possibility of writing higher-order *inductive definitions* that are natural for describing the most common recursive data structures, such as singly- or doubly-linked lists (SLLs/DLLs), trees, hash maps (lists of lists), and more complex variations thereof, such as nested and overlaid structures (e.g. lists with head and tail pointers, skip-lists, trees with linked leaves, etc.). In addition to being an appealing specification tool, SL is particularly suited for compositional reasoning about programs. Indeed, the principle of *local reasoning* allows one to verify different elements (functions, threads) of a program, operating on disjoint parts of the memory, and to combine the results a-posteriori, into succinct verification conditions.

However, the expressive power of SL comes at the price of undecidability [6]. To avoid this problem, most SL dialects used by various tools (e.g. SPACE INVADER [2], PREDATOR [9], or INFER [7]) use hard-coded predicates, describing SLLs and DLLs, for which entailments are, in general, tractable [8]. For graph structures of bounded tree width, a general decidability result was presented in [14]. Entailment in this fragment is EXPTIME-hard, as proven in [1].

In this paper, we present a novel decision procedure for a restriction of the decidable SL fragment from [14], describing recursive structures in which *all edges are local with respect to a spanning tree*. Examples of such structures include SLLs, DLLs, trees and trees with parent pointers, etc. For structures outside of this class (e.g. skip-lists or trees with linked leaves), our procedure is sound (namely, if the answer of the procedure is

F. Cassez and J.-F. Raskin (Eds.): ATVA 2014, LNCS 8837, pp. 201–218, 2014.

positive, then the entailment holds), but not complete (the answer might be negative and the entailment could still hold). In terms of program verification, such a lack of completeness in the entailment prover can lead to non-termination or false positives, but will not cause unsoundness (i.e. classify a buggy program as correct).

The method described in the paper belongs to the class of *automata-theoretic* decision techniques: We translate an entailment problem $\varphi \models \psi$ into a language inclusion problem $\mathcal{L}(A_\varphi) \subseteq \mathcal{L}(A_\psi)$ for tree automata (TA) A_φ and A_ψ that (roughly speaking) encode the sets of models of φ and ψ, respectively. Yet, a naïve translation of the inductive definitions of SL into TA encounters a *polymorphic representation* problem: the same set of structures can be defined in several different ways, and TA simply mirroring the definition will not report the entailment. For example, DLLs with selectors `next` and `prev` for the next and previous nodes, respectively, can be described by a forward unfolding of the inductive definition: $\mathtt{DLL}(\mathit{head},\mathit{prev},\mathit{tail},\mathit{next}) \equiv \exists x.\ \mathit{head} \mapsto (x,\mathit{prev}) * \mathtt{DLL}(x,\mathit{head},\mathit{tail},\mathit{next}) \mid \mathbf{emp} \wedge \mathit{head} = \mathit{tail} \wedge \mathit{prev} = \mathit{next}$, as well as by a backward unfolding of the definition: $\mathtt{DLL}_{\mathit{rev}}(\mathit{head},\mathit{prev},\mathit{tail},\mathit{next}) \equiv \exists x.\ \mathit{tail} \mapsto (\mathit{next},x) * \mathtt{DLL}_{\mathit{rev}}(\mathit{head},\mathit{prev},x,\mathit{tail}) \mid \mathbf{emp} \wedge \mathit{head} = \mathit{tail} \wedge \mathit{prev} = \mathit{next}$. Also, one can define a DLL starting with a node in the middle and unfolding backward to the left of this node and forward to the right: $\mathtt{DLL}_{\mathit{mid}}(\mathit{head},\mathit{prev},\mathit{tail},\mathit{next}) \equiv \exists x,y,z\ .\ \mathtt{DLL}(y,x,\mathit{tail},\mathit{next}) * \mathtt{DLL}_{\mathit{rev}}(\mathit{head},\mathit{prev},z,x)$. The circular entailment: $\mathtt{DLL}(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}) \models \mathtt{DLL}_{\mathit{rev}}(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}) \models \mathtt{DLL}_{\mathit{mid}}(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}) \models \mathtt{DLL}(\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d})$ holds, but a naïve structural translation to TA might not detect this fact. To bridge this gap, we define a closure operation on TA, called *canonical rotation*, which adds all possible representations of a given inductive definition, encoded as a tree automaton.

The translation from SL to TA provides also tight complexity bounds, showing that entailment in the local fragment of SL with inductive definitions is EXPTIME-complete. Moreover, we implemented our method using the VATA [17] tree automata library, which leverages from recent advances in non-deterministic language inclusion for TA [4], and obtained quite encouraging experimental results.

Related Work. Given the large body of literature on logics for describing mutable data structures, we need to restrict this section to the related work that focuses on SL [22]. The first (proof-theoretic) decidability result for SL on a restricted fragment defining only SLLs was reported in [3], which describe a co-NP algorithm. The full basic SL without recursive definitions, but with the magic wand operator was found to be undecidable when interpreted *in any memory model* [6]. A PTIME entailment procedure for SL with list predicates is given in [8]. Their method was extended to reason about nested and overlaid lists in [11]. More recently, entailments in an important SL fragment with hardcoded SLL/DLL predicates were reduced to Satisfiability Modulo Theories (SMT) problems, leveraging from recent advances in SMT technology [20,18]. The work reported in [10] deals with entailments between inductive SL formulae describing nested list structures. It uses a combination of graphs and TA to encode models of SL, but it does not deal with the problem of polymorphic representation. Recently, a decision procedure for entailments in a fragment of multi-sorted first-order logic with reachability, hard-coded trees and frame specifications, called GRIT (Graph Reachability and Inverted Trees) has been reported in [21]. Due to the restriction of the transitive closure to one function symbol (parent pointer), the expressive power of their logic, without

data constraints, is strictly lower than ours (regular properties of trees cannot be encoded in GRIT). However, GRIT can be extended with data, which has not been, so far, considered for SL.

Closer to our work on SL with user-provided *inductive definitions* is the fragment used in the tool SLEEK, which implements a semi-algorithmic entailment check, based on unfoldings and unifications [19]. Along this line of work, the theorem prover CYCLIST builds entailment proofs using a sequent calculus. Neither SLEEK nor CYCLIST are complete for a given fragment of SL, and, moreover, these tools do not address the polymorphic representation problem.

Our previous work [14] gave a general decidability result for SL with inductive definitions interpreted over graph-like structures, under several necessary restrictions, based on a reduction from SL to Monadic Second Order Logic (MSOL) on graphs of bounded tree width. Decidability of MSOL on such graphs relies on a combinatorial reduction to MSOL on trees (see [12] for a proof of Courcelle's theorem). Altogether, using the method from [14] causes a blowup of several exponentials in the size of the input problem and is unlikely to produce an effective decision procedure.

The work [1] provides a rather complete picture of complexity for the entailment in various SL fragments with inductive definitions, including EXPTIME-hardness of the decidable fragment of [14], but provides no upper bound. The EXPTIME-completeness result in this paper provides an upper bound for a fragment of *local definitions*, and strengthens the EXPTIME-hard lower bound as well, i.e. it is showed that even the entailment between local definitions is EXPTIME-hard.

2 Definitions

The set of natural numbers is denoted by $\mathbb{N}$. If $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$ and $\mathbf{y} = \langle y_1, \ldots, y_m \rangle$ are tuples, $\mathbf{x} \cdot \mathbf{y} = \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$ denotes their concatenation, $|\mathbf{x}| = n$ denotes the length of $\mathbf{x}$, and $(\mathbf{x})_i = x_i$ denotes the i-th element of $\mathbf{x}$. For a partial function $f : A \rightharpoonup B$, and $\bot \notin B$, we denote by $f(x) = \bot$ the fact that f is undefined at some point $x \in A$. The domain of f is denoted $dom(f) = \{x \in A \mid f(x) \neq \bot\}$, and the image of f is denoted as $img(f) = \{y \in B \mid \exists x \in A \,.\, f(x) = y\}$. By $f : A \rightharpoonup_{fin} B$, we denote any partial function whose domain is finite. Given two partial functions f, g defined on disjoint domains, i.e. $dom(f) \cap dom(g) = \emptyset$, we denote by $f \oplus g$ their union.

States. We consider $Var = \{x, y, z, \ldots\}$ to be a countably infinite set of *variables* and $\mathbf{nil} \in Var$ be a designated variable. Let Loc be a countably infinite set of locations and $null \in Loc$ be a designated location.

Definition 1. *A* state *is a pair $\langle s, h \rangle$ where $s : Var \rightharpoonup Loc$ is a partial function mapping pointer variables into locations such that $s(\mathbf{nil}) = null$, and $h : Loc \rightharpoonup_{fin} \mathbb{N} \rightharpoonup_{fin} Loc$ is a finite partial function such that (i) $null \notin dom(h)$ and (ii) for all $\ell \in dom(h)$ there exists $k \in \mathbb{N}$ such that $(h(\ell))(k) \neq \bot$.*

Given a state $S = \langle s, h \rangle$, s is called the *store* and h the *heap*. For any $l, l' \in Loc$, we write $\ell \xrightarrow{k}_S \ell'$ instead of $(h(\ell))(k) = \ell'$ for any $k \in \mathbb{N}$ called a *selector*. We call the triple $\ell \xrightarrow{k}_S \ell'$ an *edge* of S. When the S subscript is obvious from the context, we

sometimes omit it. Let $Img(h) = \bigcup_{\ell \in Loc} img(h(\ell))$ be the set of locations which are destinations of some edge in h. A location $\ell \in Loc$ is said to be *allocated* in $\langle s,h \rangle$ if $\ell \in dom(h)$ (i.e. it is the source of an edge). The location is called *dangling* in $\langle s,h \rangle$ if $\ell \in [img(s) \cup Img(h)] \setminus dom(h)$, i.e. it is referenced by a store variable or reachable from an allocated location in the heap, but it is not allocated in the heap itself. The set $loc(S) = img(s) \cup dom(h) \cup Img(h)$ is the set of all locations either allocated or referenced in the state S.

For any two states $S_1 = \langle s_1,h_1 \rangle$ and $S_2 = \langle s_2,h_2 \rangle$ such that (i) s_1 and s_2 agree on the evaluation of common variables ($\forall x \in dom(s_1) \cap dom(s_2) \; . \; s_1(x) = s_2(x)$) and (ii) h_1 and h_2 have disjoint domains ($dom(h_1) \cap dom(h_2) = \emptyset$), we denote by $S_1 \uplus S_2 = \langle s_1 \cup s_2, h_1 \oplus h_2 \rangle$ the *disjoint union* of S_1 and S_2. The disjoint union is undefined if one of the above conditions does not hold.

Trees and Tree Automata. Let Σ be a countable alphabet and $\mathbb{N}^*$ be the set of sequences of natural numbers. Let $\varepsilon \in \mathbb{N}^*$ denote the empty sequence and $p.q$ denote the concatenation of two sequences $p,q \in \mathbb{N}^*$. We say that p is a *prefix* of q if $q = p.q'$ for some $q' \in \mathbb{N}^*$. A set $X \subseteq \mathbb{N}^*$ is *prefix-closed* iff $p \in X \Rightarrow q \in X$ for each prefix q of p.

A *tree* t over Σ is a finite partial function $t : \mathbb{N}^* \rightharpoonup_{fin} \Sigma$ such that $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$ and, for each $p \in dom(t)$ and $i \in \mathbb{N}$, we have $t(p.i) \neq \bot$ only if $t(p.j) \neq \bot$, for all $0 \leq j < i$. The sequences $p \in dom(t)$ are called *positions* in the following. Given two positions $p,q \in dom(t)$, we say that q is the i-th successor (child) of p if $q = p.i$, for some $i \in \mathbb{N}$. We denote by $\mathcal{D}(t) = \{-1,0,\ldots,N\}$ the *direction alphabet* of t, where $N = \max\{i \in \mathbb{N} \mid \exists p \in \mathbb{N}^* \; . \; p.i \in dom(t)\}$, and we let $\mathcal{D}_+(t) = \mathcal{D}(t) \setminus \{-1\}$. By convention, we have $(p.i).(-1) = p$, for all $p \in \mathbb{N}^*$ and $i \in \mathcal{D}_+(t)$. Given a tree t and a position $p \in dom(t)$, we define the *arity* of the position p as $\#_t(p) = \max\{d \in \mathcal{D}_+(t) \mid p.d \in dom(t)\} + 1$.

A (finite, non-deterministic, bottom-up) *tree automaton* (abbreviated as TA in the following) is a quadruple $A = \langle Q, \Sigma, \Delta, F \rangle$, where Σ is a finite alphabet, Q is a finite set of *states*, $F \subseteq Q$ is a set of *final states*, Σ is an alphabet, and Δ is a set of *transition rules* of the form $\sigma(q_1,\ldots,q_n) \rightarrow q$, for $\sigma \in \Sigma$, and $q,q_1,\ldots,q_n \in Q$. Given a tree automaton $A = \langle Q, \Sigma, \Delta, F \rangle$, for each rule $\rho = (\sigma(q_1,\ldots,q_n) \rightarrow q)$, we define its size as $|\rho| = n+1$. The size of the tree automaton is $|A| = \sum_{\rho \in \Delta} |\rho|$. A *run* of A over a tree $t : \mathbb{N}^* \rightharpoonup_{fin} \Sigma$ is a function $\pi : dom(t) \rightarrow Q$ such that, for each node $p \in dom(t)$, where $q = \pi(p)$, if $q_i = \pi(p.i)$ for $1 \leq i \leq n$, then Δ has a rule $(t(p))(q_1,\ldots,q_n) \rightarrow q$. We write $t \xRightarrow{\pi} q$ to denote that π is a run of 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* of A is defined as $\mathcal{L}(A) = \{t \mid \exists q \in F, \; t \Longrightarrow q\}$.

2.1 Separation Logic

The syntax of *basic formulae* of Separation Logic (SL) is given below:

$$
\begin{array}{l}
\alpha \; \in \; Var \setminus \{\mathbf{nil}\}; \; x \; \in \; Var; \\
\Pi ::= \alpha = x \mid \Pi_1 \wedge \Pi_2 \\
\Sigma \; ::= \mathbf{emp} \mid \alpha \mapsto (x_1,\ldots,x_n) \mid \Sigma_1 * \Sigma_2 \text{ , for some } n > 0 \\
\varphi \; ::= \Sigma \wedge \Pi \mid \exists x \; . \; \varphi
\end{array}
$$

A formula of the form $\bigwedge_{i=1}^{n} \alpha_i = x_i$ defined by the Π nonterminal in the syntax above is said to be *pure*. The atomic proposition **emp**, or any formula of the form $\bigstar_{i=1}^{k} \alpha_i \mapsto$

$(x_{i,1},\ldots,x_{i,n_i})$, for some $k>0$, is said to be *spatial*. A variable x is said to be *free* in φ if it does not occur under the scope of any existential quantifier. We denote by $FV(\varphi)$ the set of free variables. A variable $\alpha \in FV(\Sigma)\setminus\{\mathbf{nil}\}$ is said to be *allocated* (respectively, *referenced*) in a spatial formula Σ if it occurs on the left-hand (respectively, right-hand) side of a proposition $\alpha \mapsto (x_1,\ldots,x_n)$ of Σ.

In the following, we shall use two equality relations. The *syntactic equality*, denoted $\sigma \equiv \varsigma$, means that σ and ς are the same syntactic object (formula, variable, tuple of variables, etc.). On the other hand, by writing $x =_\Pi y$, for two variables $x,y \in Var$ and a pure formula Π, we mean that the equality of the values of x and y is implied by Π.

A system of *inductive definitions* (inductive system) $\mathcal{P}$ is a set of rules of the form

$$\left\{ P_i(x_{i,1},\ldots,x_{i,n_i}) \equiv |_{j=1}^{m_i} R_{i,j}(x_{i,1},\ldots,x_{i,n_i}) \right\}_{i=1}^{k} \tag{1}$$

where $\{P_1,\ldots,P_k\}$ is a set of *predicates*, $x_{i,1},\ldots,x_{i,n_i}$ are called *formal parameters*, and the formulae $R_{i,j}$ are called the *rules* of P_i. Each rule is of the form $R_{i,j}(\mathbf{x}) \equiv \exists \mathbf{z} \,.\, \Sigma * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m) \wedge \Pi$, where $\mathbf{x} \cap \mathbf{z} = \emptyset$, and the following holds:

1. $\Sigma \not\equiv \mathbf{emp}$ is a non-empty spatial formula[1], called the *head* of $R_{i,j}$.
2. $P_{i_1}(\mathbf{y}_1),\ldots,P_{i_m}(\mathbf{y}_m)$ is a tuple of *predicate occurrences*, called the *tail* of $R_{i,j}$, where $|\mathbf{y}_j| = n_{i_j}$, for all $1 \leq j \leq m$.
3. Π is a pure formula, restricted such that, for all formal parameters $\beta \in \mathbf{x}$, we allow only equalities of the form $\alpha =_\Pi \beta$, where α is allocated in Σ.[2]
4. for all $1 \leq r,s \leq m$, if $x_{i,k} \in \mathbf{y}_r$, $x_{i,l} \in \mathbf{y}_s$, and $x_{i,k} =_\Pi x_{i,l}$, for some $1 \leq k,l \leq n_i$, then $r = s$; a formal parameter of a rule cannot be passed to two or more subsequent occurrences of predicates in that rule.[3]

The size of a rule R is denoted by $|R|$ and defined inductively as follows: $|\alpha = x| = 1$, $|\mathbf{emp}| = 1$, $|\alpha \mapsto (x_1,\ldots,x_n)| = n+1$, $|\varphi \bullet \psi| = |\varphi| + |\psi|$, $|\exists x \,.\, \varphi| = |\varphi| + 1$, and $|P(x_1,\ldots,x_n)| = n$. Here, $\alpha \in Var \setminus \{\mathbf{nil}\}$, $x,x_1,\ldots,x_n \in Var$, and $\bullet \in \{*,\wedge\}$. The size of an inductive system (1) is defined as $|\mathcal{P}| = \sum_{i=1}^{k} \sum_{j=1}^{m_i} |R_{i,j}|$. A *rooted system* $\langle \mathcal{P}, P_i \rangle$ is an inductive system $\mathcal{P}$ with a designated predicate $P_i \in \mathcal{P}$.

Example 1. To illustrate the use of inductive definitions (with the above restrictions), we first show how to define a predicate $\mathtt{DLL}(hd,p,tl,n)$ describing doubly-linked lists of length at least one. As depicted on the top of Fig. 1, the formal parameter *hd* points to the first allocated node of such a list, *p* to the node pointed to by the *prev* selector of *hd*, *tl* to the last node of the list (possibly equal to *hd*), and *n* to the node pointed to by the *next* selector from *tl*. This predicate can be defined as follows: $\mathtt{DLL}(hd,p,tl,n) \equiv hd \mapsto (n,p) \wedge hd = tl \mid \exists x.\ hd \mapsto (x,p) * \mathtt{DLL}(x,hd,tl,n)$.

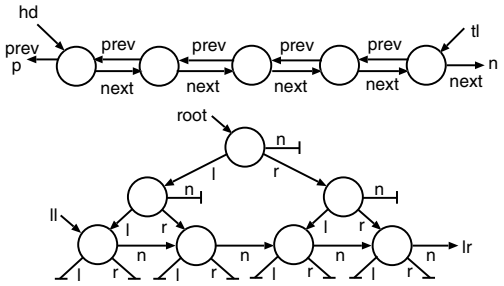


Fig. 1. Top: A DLL. Bottom: A TLL

[1] In practice, we allow frontier or root rules to have **emp**ty heads.

[2] This restriction can be lifted at the expense of an exponential blowup in the size of the TA.

[3] The restriction can be lifted by testing double allocation as in [14] (with an exponential cost).

Another example is the predicate $\mathtt{TLL}(r,ll,lr)$ describing binary trees with linked leaves whose root is pointed to by the formal parameter r, the left-most leaf is pointed to by ll, and the right-most leaf points to lr as shown in the bottom of Fig. 1: $\mathtt{TLL}(r,ll,lr) \equiv r \mapsto (\mathbf{nil},\mathbf{nil},lr) \ \wedge \ r = ll \mid \exists x,y,z.\ r \mapsto (x,y,\mathbf{nil}) * \mathtt{TLL}(x,ll,z) * \mathtt{TLL}(y,z,lr)$. ■

The semantics of SL is given by the *model relation* $\models$, defined inductively, on the structure of formulae, as follows:

$$
\begin{array}{lll}
S \models \mathbf{emp} & \iff & dom(h) = \emptyset \\
S \models \alpha \mapsto (x_1,\ldots,x_n) & \iff & s = \{(\alpha,\ell_0),(x_1,\ell_1),\ldots,(x_n,\ell_n)\} \text{ and} \\
 & & h = \{\langle \ell_0, \lambda i \ .\ \text{if } 1 \leq i \leq n \text{ then } \ell_i \text{ else } \bot \rangle\} \\
 & & \text{for some } \ell_0,\ell_1,\ldots,\ell_n \in Loc \\
S \models \varphi_1 * \varphi_2 & \iff & S_1 \models \varphi_1 \text{ and } S_2 \models \varphi_2 \text{ for some } S_1,S_2 : S_1 \uplus S_2 = S \\
S \models \exists x \ .\ \varphi & \iff & \langle s[x \leftarrow \ell], h \rangle \models \varphi \text{ for some } \ell \in Loc \\
S \models P_i(x_{i,1},\ldots,x_{i,n_i}) & \iff & S \models R_{i,j}(x_{i,1},\ldots,x_{i,n_i}), \text{ for some } 1 \leq j \leq m_i, \text{ in (1)}
\end{array}
$$

The semantics of $=$ and $\wedge$ are classical for first order logic. Note that we adopt here the *strict semantics*, in which a points-to relation $\alpha \mapsto (x_1,\ldots,x_n)$ holds in a state consisting of a single cell pointed to by α that has exactly n outgoing edges $s(\alpha) \xrightarrow{k}_S s(x_k)$, $1 \leq k \leq n$, leading either towards the single allocated location $s(\alpha)$ (if $s(x_k) = s(\alpha)$) or towards dangling locations (if $s(x_k) \neq s(\alpha)$). The empty heap is specified by **emp**.

A state S is a model of a predicate P_i iff it is a model of one of its rules $R_{i,j}$. For a state S that is a model of $R_{i,j}$, the inductive definition of the semantics implies existence of a finite *unfolding tree*: this is a tree labeled with rules of the system in such a way that, whenever a node is labeled by a rule with a tail $P_{i_1}(\mathbf{y}_1),\ldots,P_{i_m}(\mathbf{y}_m)$, it has exactly m children such that the j-th child, for $1 \leq j \leq m$, is labeled with a rule of P_{i_j} (see the middle part of Fig. 2—a formal definition is given in [16].

Given an inductive system $\mathcal{P}$, predicates $P_i(x_1,\ldots,x_n)$ and $P_j(y_1,\ldots,y_n)$ of $\mathcal{P}$ with the same number of formal parameters n, and a tuple of variables $\mathbf{x}$ where $|\mathbf{x}| = n$, the *entailment problem* is defined as follows: $P_i(\mathbf{x}) \models_{\mathcal{P}} P_j(\mathbf{x}) \ : \ \forall S \ .\ S \models P_i(\mathbf{x}) \Rightarrow S \models P_j(\mathbf{x})$.

2.2 Connectivity, Spanning Trees and Local States

In this section, we define two conditions ensuring that entailments in the restricted SL fragment can be decided effectively. The notion of a *spanning tree* is central for these definitions. Informally, a state S has a spanning tree t if all allocated locations of S can be placed in t such that there is always an edge in S in between every two locations placed in a parent-child pair of positions (see Fig. 2 for two spanning trees).

Definition 2. *Given a state $S = \langle s,h \rangle$, a* spanning tree *of S is a bijective tree $t : \mathbb{N}^* \to dom(h)$ such that $\forall p \in dom(t) \forall d \in \mathcal{D}_+(t) \ .\ p.d \in dom(t) \Rightarrow \exists k \in \mathbb{N} \ .\ t(p) \xrightarrow{k}_S t(p.d)$.*

Given an inductive system $\mathcal{P}$, let $S = \langle s,h \rangle$ be a state and $P_i \in \mathcal{P}$ be an inductive definition such that $S \models P_i$. Our first restriction, called *connectivity* (Def. 3), ensures that the unfolding tree of the definition of P_i is also a spanning tree of S (cf. Fig. 2, middle). In other words, each location $\ell \in dom(h)$ is created by an atomic proposition of the form $\alpha \mapsto (x_1,\ldots,x_n)$ from the unfolding tree of the definition P_i, and, moreover,

by Def. 2, there exists an edge $\ell \xrightarrow{k}_S \ell'$ for any parent-child pair of positions in this tree (cf. the `next` edges in Fig. 2).

For a basic quantifier-free SL formula $\varphi \equiv \Sigma \wedge \Pi$ and two variables $x, y \in FV(\varphi)$, we say that y is φ-*reachable* from x iff there is a sequence $x =_\Pi \alpha_0, \ldots, \alpha_m =_\Pi y$, for some $m \geq 0$, such that, for each $0 \leq i < m$, $\alpha_i \mapsto (\beta_{i,1}, \ldots, \beta_{i,p_i})$ is an atomic proposition in Σ, and $\beta_{i,s} =_\Pi \alpha_{i+1}$, for some $1 \leq s \leq p_i$. A variable $x \in FV(\Sigma)$ is called a *root* of Σ if every variable $y \in FV(\Sigma)$ is φ-reachable from x.

Definition 3. *Given a system $\mathcal{P} = \{P_i \equiv |_{j=1}^{m_i} R_{i,j}\}_{i=1}^{n}$ of inductive definitions, a rule $R_{i,j}(x_{i,1}, \ldots, x_{i,k}) \equiv \exists \mathbf{z} \,.\, \Sigma * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m) \wedge \Pi$ of a predicate $P_i(x_{i,1}, \ldots, x_{i,k})$ is* connected *iff there exists a formal parameter $x_{i,\ell}$ of P_i, $1 \leq \ell \leq k$, such that (i) $x_{i,\ell}$ is a root of Σ and (ii) for each $j = 1, \ldots, m$, there exists $0 \leq s < |\mathbf{y}_j|$ such that $(\mathbf{y}_j)_s$ is $(\Sigma \wedge \Pi)$-reachable from $x_{i,\ell}$ and $x_{i_j,s}$ is a root of the head of each rule of P_{i_j}. The system $\mathcal{P}$ is said to be* connected *if all its rules are connected.*

For instance, the `DLL` and `TLL` systems from Ex. 1 are both connected. Our second restriction, called *locality*, ensures that every edge $\ell \xrightarrow{k}_S \ell'$, between allocated locations $\ell, \ell' \in dom(h)$, involves locations that are mapped to a parent-child pair of positions in some spanning tree of S.

Definition 4. *Let $S = \langle s, h \rangle$ be a state and $t : \mathbb{N}^* \to dom(h)$ be a spanning tree of S. An edge $\ell \xrightarrow{k}_S \ell'$ with $\ell, \ell' \in dom(h)$ is said to be* local *w.r.t. a spanning tree t iff there exist $p \in dom(t)$ and $d \in \mathcal{D}(t) \cup \{\varepsilon\}$ such that $t(p) = \ell$ and $t(p.d) = \ell'$. The tree t is a* local spanning tree *of S iff t is a spanning tree of S and S has only local edges w.r.t. t. The state S is* local *iff it has a local spanning tree.*

For instance, the `DLL` system of Ex. 1 is local, while the `TLL` system is not (e.g. the `n` edges between leaves cannot be mapped to parent-child pairs in the spanning tree that is obtained by taking the `l` and `r` edges of the TLL). In this paper, we address the locality problem by giving a sufficient condition (a syntactic check of the inductive system, prior to the generation of TA) able to decide the locality on all of the practical examples considered (Sec. 3.2). The decidability of locality of general inductive systems is an interesting open problem, considered for future research.

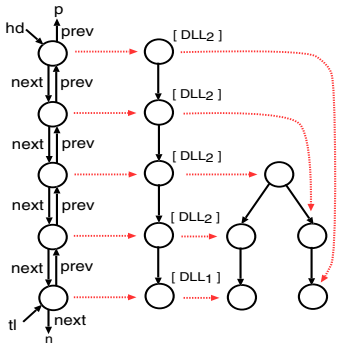


Fig. 2. Two spanning trees of a `DLL`. The middle one is an unfolding tree when labeled by $\mathtt{DLL}_1 \equiv hd \mapsto (n, p) \wedge hd = tl$ and $\mathtt{DLL}_2 \equiv \exists x.\ hd \mapsto (x, p) * \mathtt{DLL}(x, hd, tl, n)$.

Definition 5. *A system $\mathcal{P} = \{P_i(x_{i,1}, \ldots, x_{i,n_i})\}_{i=1}^{k}$ is said to be* local *if and only if each formal parameter $x_{i,j}$ of a predicate P_i is either (i) allocated in each rule of P_i and $(\mathbf{y})_j$ is referenced at each occurrence $P_i(\mathbf{y})$, or (ii) referenced in each rule of P_i and $(\mathbf{y})_j$ is allocated at each occurrence $P_i(\mathbf{y})$.*

This gives a sufficient (but not necessary) condition ensuring that any state S, such that $S \models P_i$, has a local spanning tree, if $\mathcal{P}$ is a connected local system. The condition is effective and easily implemented (see Sec. 3.2) by the translation from SL to TA.

3 From Separation Logic to Tree Automata

The first step of our entailment decision procedure is building a TA for a given inductive system. Roughly speaking, the TA we build recognizes unfolding trees of the inductive system. The alphabet of such a TA consists of small basic SL formulae describing the neighborhood of each allocated variable, together with a specification of the connections between each such formula and its parent and children in the unfolding tree. Each alphabet symbol in the TA is called a *tile*. Due to technical details related to the encoding of states as trees of SL formulae, the most space in this section is dedicated to the definition of tiles. Once the tile alphabet is defined, the states of the TA correspond naturally to the predicates of the inductive system, and the transition rules correspond to the rules of the system.

3.1 Tiles, Canonical Tiles, and Quasi-canonical Tiles

A *tile* is a tuple $T = \langle \varphi, \mathbf{x}_{-1}, \mathbf{x}_0, \ldots, \mathbf{x}_{d-1} \rangle$, for some $d \geq 0$, where φ is a basic SL formula, and each $\mathbf{x}_i$ is a tuple of pairwise distinct variables, called a *port*. We further assume that all ports contain only free variables from φ and that they are pairwise disjoint. The variables from $\mathbf{x}_{-1}$ are said to be *incoming*, the ones from $\mathbf{x}_0, \ldots, \mathbf{x}_{d-1}$ are said to be *outgoing*, and the ones from $\mathbf{par}(T) = FV(\varphi) \setminus (\mathbf{x}_{-1} \cup \ldots \cup \mathbf{x}_{d-1})$ are called *parameters*. The *arity* of a tile $T = \langle \varphi, \mathbf{x}_{-1}, \ldots, \mathbf{x}_{d-1} \rangle$ is the number of outgoing ports, denoted by $\#(T) = d$. We denote $\mathbf{form}(T) \equiv \varphi$ and $\mathbf{port}_i(T) \equiv \mathbf{x}_i$, for all $-1 \leq i < d$.

Given tiles $T_1 = \langle \varphi, \mathbf{x}_{-1}, \ldots, \mathbf{x}_{d-1} \rangle$ and $T_2 = \langle \phi, \mathbf{y}_{-1}, \ldots, \mathbf{y}_{e-1} \rangle$ such that $FV(\varphi) \cap FV(\phi) = \emptyset$, we define the *i-composition*, for some $0 \leq i < d$, such that $|\mathbf{x}_i| = |\mathbf{y}_{-1}|$: $T_1 \circledast_i T_2 = \langle \psi, \mathbf{x}_{-1}, \ldots \mathbf{x}_{i-1}, \mathbf{y}_0, \ldots, \mathbf{y}_{e-1}, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{d-1} \rangle$ where $\psi \equiv \exists \mathbf{x}_i \exists \mathbf{y}_{-1} \,.\, \varphi * \phi \wedge \mathbf{x}_i = \mathbf{y}_{-1}$.[4] For a position $q \in \mathbb{N}^*$ and a tile T, we denote by $T^{\langle q \rangle}$ the tile obtained by renaming each variable x in the ports of T by $x^{\langle q \rangle}$. A tree t labeled with tiles corresponds to a tile defined inductively, for any $p \in dom(t)$, as: $\mathcal{T}(t,p) = t(p)^{\langle p \rangle} \circledast_0 \mathcal{T}(t, p.0) \circledast_1 \mathcal{T}(t, p.1) \ \ldots \ \circledast_{\#(p)-1} \mathcal{T}(t, p.(\#_t(p)-1))$. The SL formula $\Phi(t) \equiv \mathbf{form}(\mathcal{T}(t, \varepsilon))$ is said to be the *characteristic formula* of t.

Canonical Tiles. We first define a class of tiles that encode local states (Def. 4) with respect to the underlying tile-labeled spanning trees. We denote by $T = \langle (\exists z) \ z \mapsto (y_0, \ldots, y_{m-1}) \wedge \Pi, \mathbf{x}_{-1}, \ldots, \mathbf{x}_{d-1} \rangle$ a tile whose spatial formula is either (i) $\exists z \,.\, z \mapsto (y_0, \ldots, y_{m-1})$ or (ii) $z \mapsto (y_0, \ldots, y_{m-1})$ with $z \in \mathbf{par}(T)$. A tile $T = \langle (\exists z) \ z \mapsto (y_0, \ldots, y_{m-1}) \wedge \Pi, \ \mathbf{x}_{-1}, \ldots, \mathbf{x}_{d-1} \rangle$ is said to be *canonical* if each port $\mathbf{x}_i$ can be factorized as $\mathbf{x}_i^{fw} \cdot \mathbf{x}_i^{bw}$ (distinguishing *forward* links going from the root to the leaves and *backward* links going in the opposite direction, respectively) such that:

1. $\mathbf{x}_{-1}^{bw} \equiv \langle y_{h_0}, \ldots, y_{h_k} \rangle$, for some ordered sequence $0 \leq h_0 < \ldots < h_k < m$, i.e. the backward incoming tuple consists only of variables referenced by the unique allocated variable z, ordered by the corresponding selectors.
2. For all $0 \leq i < d$, $\mathbf{x}_i^{fw} \equiv \langle y_{j_0}, \ldots, y_{j_{k_i}} \rangle$, for some ordered sequence $0 \leq j_0 < \ldots < j_{k_i} < m$. As above, each forward outgoing tuple consists of variables referenced by the unique allocated variable z, ordered by the corresponding selectors.

[4] For two tuples $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$ and $\mathbf{y} = \langle y_1, \ldots, y_k \rangle$, we write $\mathbf{x} = \mathbf{y}$ for $\bigwedge_{i=1}^{k} x_i = y_i$.

3. For all $0 \leq i, j < d$, if $(\mathbf{x}_i^{fw})_0 \equiv y_p$ and $(\mathbf{x}_j^{fw})_0 \equiv y_q$, for some $0 \leq p < q < m$ (i.e. $y_p \not\equiv y_q$), then $i < j$. This means that the forward outgoing tuples are ordered by the selectors referencing their first element.
4. $(\mathbf{x}_{-1}^{fw} \cup \mathbf{x}_0^{bw} \cup \ldots \cup \mathbf{x}_{d-1}^{bw}) \cap \{y_0, \ldots, y_{m-1}\} = \emptyset$ and $\Pi \equiv \mathbf{x}_{-1}^{fw} = z \wedge \bigwedge_{i=0}^{d-1} \mathbf{x}_i^{bw} = z$.[5]

We denote by $\mathbf{port}_i^{fw}(T)$ and $\mathbf{port}_i^{bw}(T)$ the tuples $\mathbf{x}_i^{fw}$ and $\mathbf{x}_i^{bw}$, respectively, for all $-1 \leq i < d$. The set of canonical tiles is denoted as $\mathcal{T}^c$.

Definition 6. *A tree* $t : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^c$ *is called* canonical *iff* $\#(t(p)) = \#_t(p)$ *for any* $p \in dom(t)$ *and, moreover, for each* $0 \leq i < \#_t(p)$, $|\mathbf{port}_i^{fw}(t(p))| = |\mathbf{port}_{-1}^{fw}(t(p.i))|$ *and* $|\mathbf{port}_i^{bw}(t(p))| = |\mathbf{port}_{-1}^{bw}(t(p.i))|$.

An important property of canonical trees is that each state that is a model of the characteristic formula $\Phi(t)$ of a canonical tree t (i.e. $S \models \Phi(t)$) can be uniquely described by a *local spanning tree* $u : dom(t) \to Loc$, which has the same structure as t, i.e. $dom(u) = dom(t)$. Intuitively, this is because each variable y_i, referenced in an atomic proposition $z \mapsto (y_0, \ldots, y_{m-1})$ in a canonical tile, is allocated only if it belongs to the backward part of the incoming port $\mathbf{x}_{-1}^{bw}$ or the forward part of some outgoing port $\mathbf{x}_i^{fw}$. In the first case, y_i is equal to the variable allocated by the parent tile, and in the second case, it is equal to the variable allocated by the i-th child. An immediate consequence is that any two models of $\Phi(t)$ differ only by a renaming of the allocated locations, i.e. they are identical up to isomorphism.

Example 2 (cont. of Ex. 1). To illustrate the notion of canonical trees, Fig. 3 shows two canonical trees for a given DLL. The tiles are depicted as big rectangles containing the appropriate basic formula as well as the input and output ports. In all ports, the first variable is in the forward and the second in the backward part.

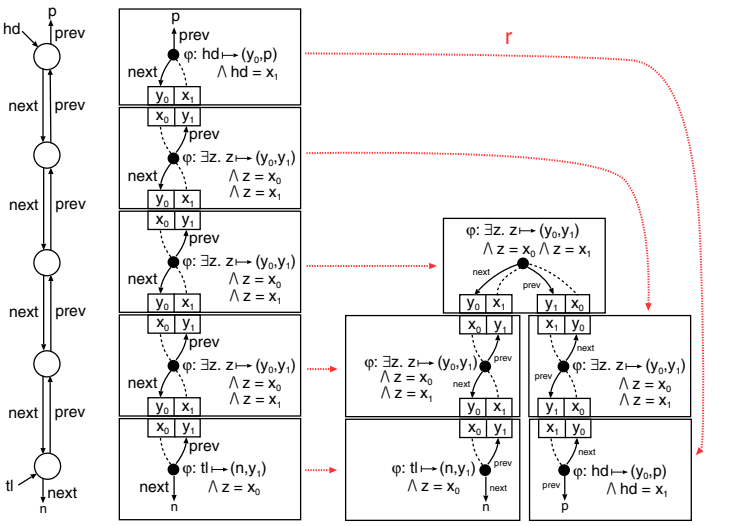


Fig. 3. The DLL from Fig. 1 with two of its canonical trees (related by a canonical rotation r) ■

Quasi-canonical tiles. We next define a class of tiles that encode non-local states in order to extend our decision procedure to handle entailments between non-local inductive systems. In addition to local edges between neighboring tiles, quasi-canonical tiles

[5] For a tuple $\mathbf{x} = \langle x_1, \ldots, x_k \rangle$, we write $\mathbf{x} = z$ for $\bigwedge_{i=1}^{k} x_i = z$.

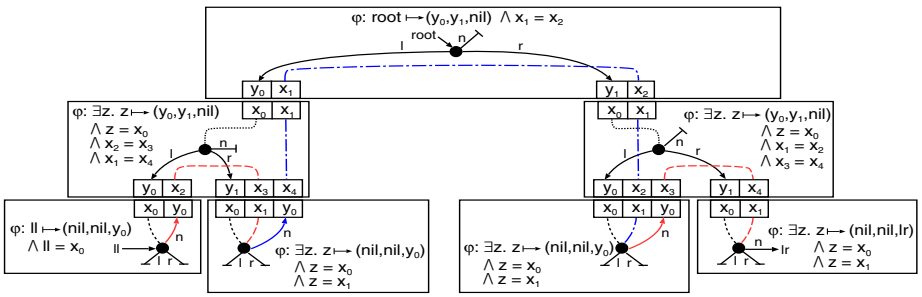


Fig. 4. A quasi-canonically tiled tree for the tree with linked leaves from Fig. 1

allow to define sequences of equalities between remote tiles. This extension is used to specify non-local edges within the state. A tile $T = \langle \varphi \wedge \Pi, \mathbf{x}_{-1}, \ldots, \mathbf{x}_{d-1} \rangle$ is said to be *quasi-canonical* if and only if each port $\mathbf{x}_i$ can be factorized as $\mathbf{x}_i^{fw} \cdot \mathbf{x}_i^{bw} \cdot \mathbf{x}_i^{eq}$, $\langle \varphi,\ \mathbf{x}_{-1}^{fw} \cdot \mathbf{x}_{-1}^{bw},\ \ldots,\ \mathbf{x}_{d-1}^{fw} \cdot \mathbf{x}_{d-1}^{bw} \rangle$ is a canonical tile, Π is pure formula, and:

1. for each $0 \leq i < |\mathbf{x}_{-1}^{eq}|$, either $(\mathbf{x}_{-1}^{eq})_i \in FV(\varphi)$ or $(\mathbf{x}_{-1}^{eq})_i =_\Pi (\mathbf{x}_k^{eq})_j$ for some unique indices $0 \leq k < d$ and $0 \leq j < |\mathbf{x}_k^{fw}|$.
2. for each $0 \leq k < d$ and each $0 \leq j < |\mathbf{x}_k^{eq}|$, either $(\mathbf{x}_k^{eq})_j \in FV(\varphi)$ or exactly one of the following holds: (i) $(\mathbf{x}_k^{eq})_j =_\Pi (\mathbf{x}_{-1}^{eq})_i$ for some unique index $0 \leq i < |\mathbf{x}_{-1}^{eq}|$ or (ii) $(\mathbf{x}_k^{eq})_j =_\Pi (\mathbf{x}_r^{eq})_s$ for some unique indices $0 \leq r < d$ and $0 \leq s < |\mathbf{x}_r^{eq}|$.
3. For any $x, y \in \bigcup_{i=-1}^{d-1} \mathbf{x}_i^{eq}$, we have $x =_\Pi y$ only in one of the cases above.

We denote $\mathbf{port}_i^{eq}(T) \equiv \mathbf{x}_i^{eq}$, for all $-1 \leq i < d$. The set of quasi-canonical tiles is denoted by $\mathcal{T}^{qc}$. The next definition of quasi-canonical trees extends Def. 6 to the case of quasi-canonical tiles.

Definition 7. *A tree* $t : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^{qc}$ *is* quasi-canonical *iff* $\#(t(p)) = \#_t(p)$ *for any* $p \in dom(t)$ *and, moreover, for each* $0 \leq i < \#_t(p)$, $|\mathbf{port}_i^{fw}(t(p))| = |\mathbf{port}_{-1}^{fw}(t(p.i))|$, $|\mathbf{port}_i^{bw}(t(p))| = |\mathbf{port}_{-1}^{bw}(t(p.i))|$, *and* $|\mathbf{port}_i^{eq}(t(p))| = |\mathbf{port}_{-1}^{eq}(t(p.i))|$.

Example 3 (cont. of Ex. 1). For an illustration of the notion of quasi-canonical trees, see Fig. 4, which shows a quasi-canonical tree for the TLL from Fig. 1. The figure uses the same notation as Fig. 3. In all the ports, the first variable is in the forward part, the backward part is empty, and the rest is the equality part. ■

3.2 Building a TA for an Inductive System

In the rest of this section, we consider that $\mathcal{P}$ is a connected inductive system (Def. 3)—our construction will detect and reject disconnected systems. Given a rooted system $\langle \mathcal{P}, P_r \rangle$, the first ingredient of our decision procedure for entailments is a procedure for building a TA that recognizes all unfolding trees of the inductive definition of P_r in the system $\mathcal{P}$. The first steps of the procedure implement a *specialization* of the rooted system with respect to a tuple $\overline{\alpha} = \langle \alpha_1, \ldots, \alpha_{n_r} \rangle$ of actual parameters for P_r, not used in $\mathcal{P}$. For space reasons, the specialization steps are described only informally here (for a detailed description of these steps, see [16]).

The first step is an elimination of existentially quantified variables that occur within equalities with formal parameters or allocated variables from all rules of $\mathcal{P}$. Second,

each rule of $\mathcal{P}$ whose head consists of more than one atomic proposition $\alpha \mapsto (x_1, \ldots, x_n)$ is split into several new rules, containing exactly one such atomic proposition. At this point, any disconnected inductive system (Def. 3) passed to the procedure is detected and rejected. The final specialization step consists in propagating the actual parameters $\overline{\alpha}$ through the rules. A formal parameter $x_{i,k}$ of a rule $R_{i,j}(x_{i,1}, \ldots, x_{i,n_i}) \equiv \exists \mathbf{z} \,.\, \Sigma * P_{i_1}(\mathbf{y}_1) * \ldots * P_{i_m}(\mathbf{y}_m) \wedge \Pi$ is *directly propagated* to some (unique) parameter of a predicate occurrence P_{i_j}, for some $1 \leq j \leq m$, if and only if $x_{i,k} \notin FV(\Sigma)$ and $x_{i,k} \equiv (\mathbf{y}_{i_j})_\ell$, for some $0 \leq \ell < |\mathbf{y}_{i_j}|$, i.e. $x_{i,k}$ is neither allocated nor pointed to by the head of the rule before being passed on to P_{i_j}. We denote direct propagation of parameters by the relation $x_{i,k} \rightsquigarrow x_{i_j,\ell}$ where $x_{i_j,\ell}$ is the formal parameter of P_{i_j} which is mapped to the occurrence of $(\mathbf{y}_{i_j})_\ell$. We say that $x_{i,k}$ is *propagated* to $x_{r,s}$ if $x_{i,k} \rightsquigarrow^* x_{r,s}$ where $\rightsquigarrow^*$ denotes the reflexive and transitive closure of the $\rightsquigarrow$ relation. Finally, we replace each variable y of $\mathcal{P}$ by the actual parameter α_j provided that $x_{r,j} \rightsquigarrow^* y$. It is not hard to show that the specialization procedure runs in time $O(|\mathcal{P}|)$, hence the size of the output system is increased by a linear factor only.

Example 4 (cont. of Ex. 1). As an example of specialization, let us consider the predicate DLL from Ex. 1, with parameters $\texttt{DLL}(\texttt{a},\texttt{b},\texttt{c},\texttt{d})$. After the parameter elimination and renaming the newly created predicates, we have a call Q_1 (without parameters) of the following inductive system:

$$Q_1() \equiv \texttt{a} \mapsto (\texttt{d},\texttt{b}) \;\wedge\; \texttt{a} = \texttt{c} \mid \exists x.\, \texttt{a} \mapsto (x,\texttt{b}) * Q_2(x,\texttt{a})$$
$$Q_2(hd,p) \equiv hd \mapsto (\texttt{d},p) \;\wedge\; hd = \texttt{c} \mid \exists x.\, hd \mapsto (x,p) * Q_2(x,hd)$$

■

We are now ready to describe the construction of a TA for a specialized rooted system $\langle \mathcal{P}, P_r \rangle$. First, for each predicate $P_j(x_{j,1}, \ldots, x_{j,n_j}) \in \mathcal{P}$, we compute several sets of parameters, called *signatures*: $\texttt{sig}_j^{fw} = \{x_{j,k} \mid x_{j,k}$ is allocated in each rule of P_j, and $(\mathbf{y})_k$ is referenced in each occurrence $P_j(\mathbf{y})$ of $P_j\}$, $\texttt{sig}_j^{bw} = \{x_{j,k} \mid x_{j,k}$ is referenced in each rule of P_j, and $(\mathbf{y})_k$ is allocated at each occurrence $P_j(\mathbf{y})$ of $P_j\}$, and, finally, $\texttt{sig}_j^{eq} = \{x_{j,1}, \ldots, x_{j,n_j}\} \setminus (\texttt{sig}_j^{fw} \cup \texttt{sig}_j^{bw})$. The signatures of an inductive system can be used to implement the *locality test* (Def. 5): the system $\mathcal{P} = \{P_1, \ldots, P_k\}$ is local if and only if $sig_i^{eq} = \emptyset$ for each $1 \leq i \leq k$.

Example 5 (cont. of Ex. 4). The signatures for the system in Ex. 4 are: $\texttt{sig}_1^{fw} = \texttt{sig}_1^{bw} = \texttt{sig}_1^{eq} = \emptyset$ and $\texttt{sig}_2^{fw} = \{hd\}, \texttt{sig}_2^{bw} = \{p\}, \texttt{sig}_2^{eq} = \emptyset$. The fact that, for each $i = 1,2$, we have $\texttt{sig}_i^{eq} = \emptyset$ implies that the DLL system is local. ■

The procedure for building a TA from a rooted system $\langle \mathcal{P}, P_r \rangle$ with actual parameters $\overline{\alpha}$ is denoted as $\text{SL2TA}(\mathcal{P}, P_r, \overline{\alpha})$ in the following. For each rule $R_{j,\ell}$ in the system, the SL2TA procedure creates a quasi-canonical tile whose incoming and outgoing ports $\mathbf{x}_i$ are factorized as $\mathbf{x}_i^{fw} \cdot \mathbf{x}_i^{bw} \cdot \mathbf{x}_i^{eq}$ according to the precomputed signatures $\texttt{sig}_j^{fw}$, $\texttt{sig}_j^{bw}$, and $\texttt{sig}_j^{eq}$, respectively. The backward part of the input port $\mathbf{x}_{-1}^{bw}$ and the forward parts of the output ports $\{\mathbf{x}_i^{fw}\}_{i \geq 0}$ are sorted according to the order of incoming selector edges from the single points-to formula which constitutes the head of the rule. The output ports $\{\mathbf{x}_i\}_{i \geq 0}$ are sorted within the tile according to the order of the selector edges

pointing to $(\mathbf{x}_i^{fw})_0$ for each $i \geq 0$. Finally, each predicate name P_i is associated with a state q_i, and for each inductive rule, the procedure creates a transition rule in the TA. The final state of the TA then corresponds to the root of the system (see Algorithm in [16]). The invariant used to prove the correctness of this construction is that whenever the TA reaches a state q_i it reads an unfolding tree whose root is labeled with a rule $R_{i,j}$ of the definition of a predicate P_i. The following lemma summarizes the TA construction:

Lemma 1. *Given a rooted system $\langle \mathcal{P}, P_r(x_{r,1},\ldots,x_{r,n_r})\rangle$ where $\mathcal{P} = \{P_i\}_{i=1}^{k}$ is a connected inductive system, $1 \leq r \leq k$, and $\overline{\alpha} = \langle \alpha_1,\ldots,\alpha_{n_i}\rangle$ is a tuple of variables not in $\mathcal{P}$, let $A = \text{SL2TA}(\mathcal{P}, P_r, \overline{\alpha})$. Then, for every state S, we have $S \models P_r(\overline{\alpha})$ iff there exists $t \in \mathcal{L}(A)$ such that $S \models \Phi(t)$. Moreover, $|A| = O(|\mathcal{P}|)$.*

Example 6 (cont. of Ex. 5). For the specialized inductive system $\mathcal{P} = \{Q_1, Q_2\}$ from Ex. 4, we obtain the TA $A = \text{SL2TA}(\mathcal{P}, Q_1, \langle \mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\rangle) = \langle \Sigma, \{q_1,q_2\}, \Delta, \{q_1\}\rangle$ where Δ is shown above. ■

$$\Delta = \left\{ \begin{array}{ll} \langle \mathtt{a} \mapsto (\mathtt{d},\mathtt{b}) \wedge \mathtt{a} = \mathtt{c}, \emptyset\rangle() \to q_1 & \langle \mathtt{a} \mapsto (x,\mathtt{b}), \emptyset, (x,\mathtt{a})\rangle(q_2) \to q_1 \\ \langle \exists hd'.hd' \mapsto (\mathtt{d},p) \wedge hd = \mathtt{c} \wedge hd' = hd, (hd,p)\rangle() & \to q_2 \\ \langle \exists hd'.hd' \mapsto (x,p) \wedge hd' = hd, (hd,p), (x,hd)\rangle(q_2) & \to q_2 \end{array} \right\}$$

4 Rotation of Tree Automata

In this section we deal with polymorphic representations of states, i.e. situations when a state can be represented by different spanning trees, with different tilings. In this section we show that, for states with local spanning trees only (Def. 4), these trees are related by a *rotation* relation.

4.1 Rotation as a Transformation of TA

We start by defining rotation as a relation on trees. Intuitively, two trees t_1 and t_2 are related by a rotation whenever we can obtain t_2 from t_1 by picking a position $p \in dom(t_1)$ and making it the root of t_2, while maintaining in t_2 all edges from t_1 (Fig. 5).

Definition 8. *Given two trees $t_1, t_2 : \mathbb{N}^* \rightharpoonup_{fin} \Sigma$ and a bijective mapping $r : dom(t_1) \to dom(t_2)$, we say that t_2 is an r-*rotation *of t_1, denoted by $t_1 \sim_r t_2$ if and only if: $\forall p \in dom(t_1) \forall d \in \mathcal{D}_+(t_1) \;:\; p.d \in dom(t_1) \Rightarrow \exists e \in \mathcal{D}(t_2)\;.\; r(p.d) = r(p).e$. We write $t_1 \sim t_2$ if there exists a bijective mapping $r : dom(t_1) \to dom(t_2)$ such that $t_1 \sim_r t_2$.*

An example of a rotation r of a tree t_1 to a tree t_2 such that $r(\varepsilon) = 2$, $r(0) = \varepsilon$, $r(1) = 20$, $r(00) = 0$, and $r(01) = 1$ is shown in Fig. 5. Note that, e.g., for $p = \varepsilon \in dom(t_1)$ and $d = 0 \in \mathcal{D}_+(t_1)$, where $p.d = \varepsilon.0 \in dom(t_1)$, we get $e = -1 \in \mathcal{D}(t_2)$, and $r(\varepsilon.0) = 2.(-1) = \varepsilon$.

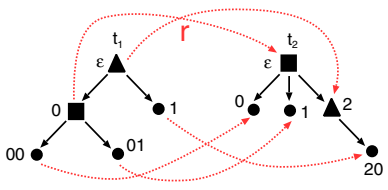


Fig. 5. An example of a rotation

In the rest of this section, we define rotation on canonical and quasi-canonical trees. These definitions are refinements of Def. 8. Namely, the change in the structure of the

tree is mirrored by a change in the tile alphabet labeling the tree in order to preserve the state which is represented by the (quasi-)canonical tree.

A *substitution* is an injective partial function $\sigma : Var \rightharpoonup_{fin} Var$. Given a basic formula φ and a substitution σ, we denote by $\varphi[\sigma]$ the result of simultaneously replacing each variable x (not necessarily free) that occurs in φ by $\sigma(x)$. For instance, if $\sigma(x) = y$, $\sigma(y) = z$, and $\sigma(z) = t$, then $(\exists x, y \,.\, x \mapsto (y,z) \wedge z = x)[\sigma] \equiv \exists y, z \,.\, y \mapsto (z,t) \ \wedge \ t = y$.

Definition 9. *Given two canonical trees $t, u : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^c$ and a bijective mapping $r : dom(t) \to dom(u)$, we say that u is a* canonical rotation *of t, denoted $t \sim_r^c u$, if and only if $t \sim_r u$ and there exists a substitution $\sigma_p : Var \rightharpoonup_{fin} Var$ for each $p \in dom(t)$ such that $\mathbf{form}(t(p))[\sigma_p] \equiv \mathbf{form}(u(r(p)))$ and, for all $0 \leq i < \#_t(p)$, there exists $j \in \mathcal{D}(u)$ such that $r(p.i) = r(p).j$ and:*

$$\mathbf{port}_i^{fw}(t(p))[\sigma_p] \equiv \textit{if } j \geq 0 \textit{ then } \mathbf{port}_j^{fw}(u(r(p))) \textit{ else } \mathbf{port}_{-1}^{bw}(u(r(p)))$$
$$\mathbf{port}_i^{bw}(t(p))[\sigma_p] \equiv \textit{if } j \geq 0 \textit{ then } \mathbf{port}_j^{bw}(u(r(p))) \textit{ else } \mathbf{port}_{-1}^{fw}(u(r(p)))$$

We write $t \sim^c u$ if there exists a mapping r such that $t \sim_r^c u$.

Example 7 (cont. of Ex. 2). The notion of canonical rotation is illustrated by the canonical rotation r relating the two canonical trees of a DLL shown in Fig. 3. In its case, the variable substitutions are simply the identity in each node. Note, in particular, that when the tile 0 of the left tree (i.e., the second one from the top) gets rotated to the tile 1 of the right tree (i.e., the right successor of the root), the input and output ports get swapped and so do their forward and backward parts. ■

The following lemma is the key for proving completeness of our entailment checking for local inductive systems: if a (local) state is a model of the characteristic formulae of two different canonical trees, then these trees must be related by canonical rotation.

Lemma 2. *Let $t : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^c$ be a canonical tree and $S = \langle s, h \rangle$ be a state such that $S \models \Phi(t)$. Then, for any canonical tree $u : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^c$, we have $S \models \Phi(u)$ iff $t \sim^c u$.*

In the following, we extend the notion of rotation to quasi-canonical trees:

Definition 10. *Given two quasi-canonical trees $t, u : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^{qc}$ and a bijective mapping $r : dom(t) \to dom(u)$, we say that u is a* quasi-canonical rotation *of t, denoted $t \sim_r^{qc} u$, if and only if $t \sim_r^c u$ and $|\mathbf{port}_i^{eq}(t(p))| = |\mathbf{port}_j^{eq}(u(r(p)))|$ for all $p \in dom(t)$ and all $0 \leq i < \#_t(p)$, $-1 \leq j < \#_t(p)$ such that $r(p.i) = r(p).j$. We write $t \sim^{qc} u$ if there exists a mapping r such that $t \sim_r^{qc} u$.*

The increase in expressivity (i.e. the possibility of defining non-local edges) comes at the cost of a loss of completeness. The following lemma generalizes the necessity direction ($\Leftarrow$) of Lemma 2 for quasi-canonical tiles. Notice that the sufficiency ($\Rightarrow$) direction does not hold in general.

Lemma 3. *Let $t, u : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^{qc}$ be quasi-canonical trees such that $t \sim^{qc} u$. For all states S, if $S \models \Phi(t)$, then $S \models \Phi(u)$.*

Algorithm 1. Rotation Closure of Quasi-canonical TA

input a quasi-canonical TA $A = \langle Q, \Sigma, \Delta, F \rangle$
output a TA A^r where:
$\mathcal{L}(A^r) = \{u : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^{qc} \mid \exists t \in \mathcal{L}(A) \,.\, u \sim^{qc} t\}$
function ROTATETA(A)
  $A^r \leftarrow A$
  assume $A^r \equiv \langle Q_r, \Sigma, \Delta_r, F_r \rangle$
  for all $\rho \in \Delta$ **do**
    assume $\rho \equiv T(q_0, \ldots, q_k) \rightarrow q$
    assume $T \equiv \langle \varphi, \mathbf{x}_{-1}, \mathbf{x}_0, \ldots, \mathbf{x}_k \rangle$
    if $\mathbf{x}_{-1} \neq \emptyset$ **or** $q \notin F$ **then**
      assume $\mathbf{x}_{-1} \equiv \mathbf{x}_{-1}^{fw} \cdot \mathbf{x}_{-1}^{bw} \cdot \mathbf{x}_{-1}^{eq}$
      if $\mathbf{x}_{-1}^{bw} \neq \emptyset$ **then**
        $Q^{rev} \leftarrow \{q^{rev} \mid q \in Q\}$
        $(Q_\rho, \Delta_\rho) \leftarrow (Q \cup Q^{rev} \cup \{q_\rho^f\}, \Delta)$
        $p \leftarrow$ POSITIONOF$(\mathbf{x}_{-1}^{bw}, \varphi)$
        $\mathbf{x}_{swap} \leftarrow \mathbf{x}_{-1}^{bw} \cdot \mathbf{x}_{-1}^{fw} \cdot \mathbf{x}_{-1}^{eq}$
        $T_{new} \leftarrow \langle \varphi, \langle\rangle, \mathbf{x}_0, \ldots, \mathbf{x}_p, \mathbf{x}_{swap}, \ldots, \mathbf{x}_k \rangle$
        $\Delta_\rho \leftarrow \Delta_\rho \cup \{T_{new}(q_0 \ldots q_p, q^{rev} \ldots q_k) \rightarrow q_\rho^f\}$
        $(\Delta_\rho, \_) \leftarrow$ ROTTR$(q, \Delta, \Delta_\rho, \emptyset, F)$
        $A_\rho \leftarrow \langle Q_\rho, \Sigma, \Delta_\rho, \{q_\rho^f\} \rangle$
        $A^r \leftarrow A^r \cup A_\delta$
  return A^r

function ROTTR$(q, \Delta, \Delta_{new}, \mathtt{V}, F)$
  $\mathtt{V} \leftarrow \mathtt{V} \cup \{q\}$
  for all $(U(s_0, \ldots, s_\ell) \rightarrow s) \in \Delta$ **do**
    for all $0 \leq j \leq \ell$ **such that** $s_j = q$ **do**
      assume $U = \langle \varphi, \mathbf{x}_{-1}, \mathbf{x}_0, \ldots, \mathbf{x}_j, \ldots, \mathbf{x}_\ell \rangle$
      assume $\mathbf{x}_j \equiv \mathbf{x}_j^{fw} \cdot \mathbf{x}_j^{bw} \cdot \mathbf{x}_j^{eq}$
      if $\mathbf{x}_{-1} = \emptyset$ **and** $s \in F$ **then**
        $\mathbf{x}_{swap} \leftarrow \mathbf{x}_j^{bw} \cdot \mathbf{x}_j^{fw} \cdot \mathbf{x}_j^{eq}$
        $U' \leftarrow \langle \varphi, \mathbf{x}_{swap}, \mathbf{x}_0, \ldots, \mathbf{x}_{j-1}, \mathbf{x}_{j+1}, \ldots, \mathbf{x}_\ell \rangle$
        $\Delta_{new} \leftarrow \Delta_{new} \cup \{U'(s_0 \ldots s_{j-1} \ldots s_\ell) \rightarrow q^{rev}\}$
      else
        $\mathbf{x}_{-1} \equiv \mathbf{x}_{-1}^{fw} \cdot \mathbf{x}_{-1}^{bw} \cdot \mathbf{x}_{-1}^{eq}$
        if $\mathbf{x}_{-1}^{bw} \neq \emptyset$ **then**
          $\mathtt{ports} \leftarrow \langle \mathbf{x}_0, \ldots, \mathbf{x}_{j-1}, \mathbf{x}_{j+1}, \ldots, \mathbf{x}_\ell \rangle$
          $\mathtt{states} \leftarrow (s_0, \ldots, s_{j-1}, s_{j+1}, \ldots, s_\ell)$
          $\mathbf{x}_{swap} \leftarrow \mathbf{x}_{-1}^{bw} \cdot \mathbf{x}_{-1}^{fw} \cdot \mathbf{x}_{-1}^{eq}$
          $p \leftarrow$ INSERTOUTPORT$(\mathbf{x}_{swap}, \mathtt{ports}, \varphi)$
          INSERTLHSSTATE$(s^{rev}, \mathtt{states}, p)$
          $U_{new} \leftarrow \langle \varphi, \mathbf{x}_j^{bw} \cdot \mathbf{x}_j^{fw} \cdot \mathbf{x}_j^{eq}, \mathtt{ports} \rangle$
          $\Delta_{new} \leftarrow \Delta_{new} \cup \{U_{new}(\mathtt{states}) \rightarrow q^{rev}\}$
          if $s \notin \mathtt{V}$ **then**
            $(\Delta_{new}, \mathtt{V}) \leftarrow$ ROTTR$(s, \Delta, \Delta_{new}, \mathtt{V}, F)$
  return $(\Delta_{new}, \mathtt{V})$

4.2 Implementing Rotation as a Transformation of TA

This section describes the algorithm that produces the closure of a quasi-canonical tree automaton (i.e. a tree automaton recognizing quasi-canonical trees only) under rotation. The result is a TA that recognizes all trees $u : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^{qc}$ such that $t \sim^{qc} u$ for some tree t recognized by the input TA $A = \langle Q, \Sigma, \Delta, F \rangle$. Algorithm 1 (the ROTATETA procedure) describes the rotation closure whose result is a language-theoretic union of A and the TA A_ρ, one for each rule ρ of A. The idea behind the construction of $A_\rho = \langle Q_\rho, \Sigma, \Delta_\rho, \{q_\rho^f\} \rangle$ can be understood by considering a tree $t \in \mathcal{L}(A)$, a run $\pi : dom(t) \rightarrow Q$, and a position $p \in dom(t)$, which is labeled with the right hand side of the rule $\rho = T(q_1, \ldots, q_k) \rightarrow q$ of A. Then $\mathcal{L}(A_\rho)$ will contain the rotated tree u, i.e. $t \sim_r^{qc} u$, where the significant position p is mapped into the root of u by the rotation function r, i.e. $r(p) = \varepsilon$. To this end, we introduce a new rule $T_{new}(q_0, \ldots, q^{rev}, \ldots, q_k) \rightarrow q_\rho^f$ where the tile T_{new} mirrors the change in the structure of T at position p, and $q^{rev} \in Q_\rho$ is a fresh state corresponding to q. The construction of A_ρ continues recursively (procedure ROTTR), by considering every rule of A that has q on the left hand side: $U(q'_1, \ldots, q, \ldots, q'_\ell) \rightarrow s$. This rule is changed by swapping the roles of q and s and producing a rule $U_{new}(q'_1, \ldots, s^{rev}, \ldots q'_\ell) \rightarrow q^{rev}$ where U_{new} mirrors the change in the structure of U. Intuitively, the states $\{q^{rev} | q \in Q\}$ mark the unique path from the root of u to $r(\varepsilon) \in dom(u)$. The recursion stops when either (i) s is a final state of A, (ii) The tile U does not specify a forward edge in the direction marked by q, or (iii) all states of A have been visited.

Lemma 4. *Let $A = \langle Q, \mathcal{T}^{qc}, \Delta, F \rangle$ be a TA, and $A^r =$ ROTATETA(A) be the TA defining the rotation closure of A. Then $\mathcal{L}(A^r) = \{u \mid u : \mathbb{N}^* \rightharpoonup_{fin} \mathcal{T}^{qc},\ \exists t \in \mathcal{L}(A) \,.\, u \sim^{qc} t\}$. Moreover, $|A^r| = O(|A|^2)$.*

The main result of this paper is given by the following theorem. The entailment problem for inductive systems is reduced, in polynomial time, to a language inclusion problem for tree automata. The inclusion test is always sound (if the answer is yes, the entailment holds), and complete, if the right-hand side is a local system (Def. 4).

Theorem 1. *Let* $\mathcal{P} = \left\{ P_i \equiv |_{j=1}^{m_i} R_{i,j} \right\}_{i=1}^{k}$ *be a connected inductive system. Then, for any two predicates* $P_i(x_{i,1}, \ldots, x_{i,n_i})$ *and* $P_j(x_{j,1}, \ldots, x_{j,n_j})$ *of* $\mathcal{P}$ *such that* $n_i = n_j$*, and for any tuple of variables* $\overline{\alpha} = \langle \alpha_1, \ldots, \alpha_{n_i} \rangle$ *not used in* $\mathcal{P}$*, the following holds for* $A_1 = \text{SL2TA}(\mathcal{P}, P_i, \overline{\alpha})$ *and* $A_2 = \text{SL2TA}(\mathcal{P}, P_j, \overline{\alpha})$*:*

- **(Soundness)** $P_i(\overline{\alpha}) \models_{\mathcal{P}} P_j(\overline{\alpha})$ *if* $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2^r)$ *and*
- **(Completness)** $P_i(\overline{\alpha}) \models_{\mathcal{P}} P_j(\overline{\alpha})$ *only if* $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2^r)$ *provided* $\langle \mathcal{P}, P_j \rangle$ *is local.*

Example 8 (cont. of Ex. 6). When applied on the tree automaton A, the operation of rotation closure produces the tree automaton $A^r = \langle \Sigma, \{q_1, q_2, q_2^{rev}, q_{fin}\}, \Delta, \{q_1, q_{fin}\} \rangle$ where Δ is shown above. ■

$$\Delta = \left\{ \begin{array}{ll}
\langle \mathtt{a} \mapsto (\mathtt{b},\mathtt{d}) \wedge \mathtt{a} = \mathtt{c}, \emptyset \rangle() \rightarrow q_1 \quad \langle \mathtt{a} \mapsto (x,\mathtt{b}), \emptyset, (x,\mathtt{a}) \rangle(q_2) & \rightarrow q_1 \\
\langle \exists hd'.hd' \mapsto (\mathtt{d},p) \wedge hd = \mathtt{c} \wedge hd' = hd, (hd,p) \rangle() & \rightarrow q_2 \\
\langle \exists hd'.hd' \mapsto (x,p) \wedge hd' = hd, (hd,p), (x,hd) \rangle(q_2) & \rightarrow q_2 \\
\langle \exists hd'.hd' \mapsto (d,p) \wedge hd = c \wedge hd' = hd, \emptyset, (p,hd) \rangle(q_2^{rev}) & \rightarrow q_{fin} \\
\langle \mathtt{a} \mapsto (x,\mathtt{b}), (\mathtt{a},x) \rangle() & \rightarrow q_2^{rev} \\
\langle \exists hd'.hd' \mapsto (x,p) \wedge hd' = hd, (hd,x), (p,hd) \rangle(q_2^{rev}) & \rightarrow q_2^{rev} \\
\langle \exists hd'.hd' \mapsto (x,p) \wedge hd' = hd, \emptyset, (x,hd), (p,hd) \rangle(q_2, q_2^{rev}) & \rightarrow q_{fin}
\end{array} \right\}$$

5 Complexity

In this section, we provide tight complexity bounds for the entailment problem in the fragment of SL with inductive definitions under consideration, i.e., with the *connectivity* and *locality* restrictions. The first result shows the need for *connectivity* within the system: allowing disconnected rules leads to undecidability of the entailment problem. As a remark, the general undecidability of entailments for SL with inductive definitions has already been proven in [1]. Our proof stresses the fact that undecidability occurs due the lack of connectivity within some rules.

Theorem 2. *Entailment is undecidable for inductive systems with disconnected rules.*

The second result of this section provides tight complexity bounds for the entailment problem for local connected systems. We must point out that EXPTIME-hardness of entailments in the fragment of [14] was already proved in [1]. The result below is stronger since the fragment under consideration is a restriction of the fragment from [14] obtained by applying the locality condition.

Theorem 3. *Entailment is EXPTIME-complete for local connected inductive systems.*

6 Experiments

We implemented a prototype tool called SLIDE (Separation Logic with Inductive DEfinitions) [15] that takes as input two rooted systems $\langle \mathcal{P}_{lhs}, P_{lhs} \rangle$ and $\langle \mathcal{P}_{rhs}, P_{rhs} \rangle$ and tests the validity of the entailment $P_{lhs} \models_{\mathcal{P}_{lhs} \cup \mathcal{P}_{rhs}} P_{rhs}$. Table 1 lists the entailment queries on which we tried out our tool; all examples are public and available on the web [15]. The upper part of the table contains local systems, whereas the bottom part contains

Table 1. Experimental results. The upper table contains local systems, while the lower table non-local ones. Sizes of initial TA (col. 3,4) and rotated TA (col. 5) are in numbers of states/transitions.

Entailment $LHS \models RHS$	Answer	$\|A_{lhs}\|$	$\|A_{rhs}\|$	$\|A^r_{rhs}\|$
$\mathtt{DLL}(a,\mathbf{nil},c,\mathbf{nil}) \models \mathtt{DLL}_{rev}(a,\mathbf{nil},c,\mathbf{nil})$	True	2/4	2/4	5/8
$\mathtt{DLL}_{rev}(a,\mathbf{nil},c,\mathbf{nil}) \models \mathtt{DLL}_{mid}(a,\mathbf{nil},c,\mathbf{nil})$	True	2/4	4/8	12/18
$\mathtt{DLL}_{mid}(a,\mathbf{nil},c,\mathbf{nil}) \models \mathtt{DLL}(a,\mathbf{nil},c,\mathbf{nil})$	True	4/8	2/4	5/8
$\exists x,n,b.\ x \mapsto (n,b) * \mathtt{DLL}_{rev}(a,\mathbf{nil},b,x) * \mathtt{DLL}(n,x,c,\mathbf{nil}) \models \mathtt{DLL}(a,\mathbf{nil},c,\mathbf{nil})$	True	3/5	2/4	5/8
$\mathtt{DLL}(a,\mathbf{nil},c,\mathbf{nil}) \models \exists x,n,b.\ x \mapsto (n,b) * \mathtt{DLL}_{rev}(a,\mathbf{nil},b,x) * \mathtt{DLL}(n,x,c,\mathbf{nil})$	False	2/4	3/5	9/13
$\exists y,a.\ x \mapsto (y,\mathbf{nil}) * y \mapsto (a,x) * \mathtt{DLL}(a,y,c,\mathbf{nil}) \models \mathtt{DLL}(x,\mathbf{nil},c,\mathbf{nil})$	True	3/4	2/4	5/8
$\mathtt{DLL}(x,\mathbf{nil},c,\mathbf{nil}) \models \exists y,a.\ x \mapsto (\mathbf{nil},y) * y \mapsto (a,x) * \mathtt{DLL}(a,y,c,\mathbf{nil})$	False	2/4	3/4	8/10
$\exists x,b.\mathtt{DLL}(x,b,c,\mathbf{nil}) * \mathtt{DLL}_{rev}(a,\mathbf{nil},b,x) \models \mathtt{DLL}(a,\mathbf{nil},c,\mathbf{nil})$	True	3/6	2/4	5/8
$\mathtt{DLL}(a,\mathbf{nil},c,\mathbf{nil}) \models \mathtt{DLL}_{0+}(a,\mathbf{nil},c,\mathbf{nil})$	True	2/4	2/4	5/8
$\mathtt{TREE}_{pp}(a,\mathbf{nil}) \models \mathtt{TREE}^{rev}_{pp}(a,\mathbf{nil})$	True	2/4	3/8	6/11
$\mathtt{TREE}^{rev}_{pp}(a,\mathbf{nil}) \models \mathtt{TREE}_{pp}(a,\mathbf{nil})$	True	3/8	2/4	5/10
$\mathtt{TLL}_{pp}(a,\mathbf{nil},c,\mathbf{nil}) \models \mathtt{TLL}^{rev}_{pp}(a,\mathbf{nil},c,\mathbf{nil})$	True	4/8	4/8	13/22
$\mathtt{TLL}^{rev}_{pp}(a,\mathbf{nil},c,\mathbf{nil}) \models \mathtt{TLL}_{pp}(a,\mathbf{nil},c,\mathbf{nil})$	True	4/8	4/8	13/22
$\exists l,r,z.\ a \mapsto (l,r,\mathbf{nil},\mathbf{nil}) * \mathtt{TLL}(l,c,z) * \mathtt{TLL}(r,z,\mathbf{nil}) \models \mathtt{TLL}(a,c,\mathbf{nil})$	True	4/7	4/8	13/22
$\mathtt{TLL}(a,c,\mathbf{nil}) \models \exists l,r,z.\ a \mapsto (l,r,\mathbf{nil},\mathbf{nil}) * \mathtt{TLL}(l,c,z) * \mathtt{TLL}(r,z,\mathbf{nil})$	False	4/8	4/7	13/21

non-local systems. Apart from the DLL and TLL predicates from Sect. 2.1, the considered entailment queries contain the following predicates: $\mathtt{DLL}_{rev}$ (resp. $\mathtt{DLL}_{mid}$) that encodes a DLL from the end (resp. middle), $\mathtt{DLL}_{0+}$ that encodes a possibly empty DLL, $\mathtt{TREE}_{pp}$ encoding trees with parent pointers, $\mathtt{TREE}^{rev}_{pp}$ that encodes trees with parent pointers defined starting with an arbitrary leaf, $\mathtt{TLL}_{pp}$ encoding TLLs with parent pointers, and $\mathtt{TLL}^{rev}_{pp}$ which encodes TLLs with parent pointers starting from their leftmost leaf. Columns $|A_{lhs}|$, $|A_{rhs}|$, and $|A^r_{rhs}|$ of Table 1 provide information about the number of states/transitions of the respective TA. The tool answered all queries correctly (despite the incompleteness for non-local systems), and the running times were all under 1 sec. on a standard PC (Intel Core2 CPU, 3GHz, 4GB RAM).

We also compared the SLIDE tool to the CYCLIST [5] theorem prover on the examples from the CYCLIST distribution [13]. Both tools run in less than 1 sec. on the examples from their common fragment of SL. CYCLIST does not handle examples where rotation is needed, while SLIDE fails on examples that generate an unbounded number of dangling pointers and are outside of the decidable fragment of [14].

7 Conclusion

We presented a novel decision procedure for the entailment problem in a non-trivial subset of SL with inductive predicates, which deals with the problem that the same recursive structure may be represented differently, when viewed from different entry points. To this end, we use a special operation, which closes a given TA representation w.r.t. the rotations of its spanning trees. Our procedure is sound and complete for inductive systems with local edges. We have implemented a prototype tool which we tested through a number of non-trivial experiments, with encouraging results.

Acknowledgment. This work was supported by the Czech Science Foundation under the project 14-11384S, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal BUT projects FIT-S-12-1 and FIT-S-14-2486.

References

1. Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M., Ouaknine, J.: Foundations for decision problems in separation logic with general inductive predicates. In: Muscholl, A. (ed.) FOSSACS 2014. LNCS, vol. 8412, pp. 411–425. Springer, Heidelberg (2014)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
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. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 350–367. Springer, Heidelberg (2012)
6. Brotherston, J., Kanovich, M.: Undecidability of propositional separation logic and its neighbours. In: Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, pp. 130–139 (2010)
7. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011)
8. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011)
9. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
10. Enea, C., Lengál, O., Sighireanu, M., Vojnar, T.: Compositional Entailment Checking for a Fragment of Separation Logic. Technical Report FIT-TR-2014-01, FIT, Brno University of Technology (2014)
11. Enea, C., Saveluc, V., Sighireanu, M.: Compositional invariant checking for overlaid and nested linked lists. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 129–148. Springer, Heidelberg (2013)
12. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer-Verlag New York, Inc. (2006)
13. Gorogiannis, N.: Cyclist: a cyclic theorem prover framework, `https://github.com/ngorogiannis/cyclist/`
14. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 21–38. Springer, Heidelberg (2013)
15. Iosif, R., Rogalewicz, A., Vojnar, T.: Slide: Separation logic with inductive definitions, `http://www.fit.vutbr.cz/research/groups/verifit/tools/slide/`
16. Iosif, R., Rogalewicz, A., Vojnar, T.: Deciding entailments in inductive separation logic with tree automata. CoRR, abs/1402.2127 (2014)
17. Lengal, O., Simacek, J., Vojnar, T.: Vata: a tree automata library, `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`

18. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic modulo theories. In: Shan, C.-C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 90–106. Springer, Heidelberg (2013)
19. Nguyen, H.H., Chin, W.-N.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008)
20. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 773–789. Springer, Heidelberg (2013)
21. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 711–728. Springer, Heidelberg (2014)
22. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS 2002. IEEE CS Press (2002)

Appendix G

Abstraction Refinement and Antichains for Trace Inclusion of Infinite State Systems

Abstraction Refinement and Antichains for Trace Inclusion of Infinite State Systems

Radu Iosif[1], Adam Rogalewicz[2], and Tomáš Vojnar[2]

[1] University Grenoble Alpes, CNRS, VERIMAG, Grenoble, France ⋆
[2] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic⋆⋆

Abstract. A *data automaton* is a finite automaton equipped with variables (counters or registers) ranging over infinite data domains. A trace of a data automaton is an alternating sequence of alphabet symbols and values taken by the counters during an execution of the automaton. The problem addressed in this paper is the inclusion between the sets of traces (data languages) recognized by such automata. Since the problem is undecidable in general, we give a semi-algorithm based on abstraction refinement, which is proved to be sound and complete modulo termination. Due to the undecidability of the trace inclusion problem, our procedure is not guaranteed to terminate. We have implemented our technique in a prototype tool and show promising results on several non-trivial examples.

1 Introduction

In this paper, we address a *trace inclusion* problem for infinite-state systems. Given (i) a network of *data automata* $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ that communicate via a set of shared variables $\mathbf{x}_\mathcal{A}$, ranging over an infinite data domain, and a set of input events $\Sigma_\mathcal{A}$, and (ii) a data automaton B whose set of variables $\mathbf{x}_B$ is a subset of $\mathbf{x}_\mathcal{A}$, does the set of (finite) traces of B contain the traces of $\mathcal{A}$? Here, by a *trace*, we understand an alternating sequence of valuations of the variables from the set $\mathbf{x}_B$ and input events from the set $\Sigma_\mathcal{A} \cap \Sigma_B$, starting and ending with a valuation. Typically, the network of automata $\mathcal{A}$ is an implementation of a concurrent system and B is a specification of the set of good behaviors of the system.

Consider, for instance, the network $\langle A_1, \ldots, A_N \rangle$ of data automata equipped with the integer-valued variables x and v shown in Fig. 1 (left). The automata synchronize on the **init** symbol and interleave their $\mathbf{a}_{1,\ldots,N}$ actions. Each automaton A_i increases the shared variable x and writes its identifier i into the shared variable v as long as the value of x is in the interval $[(i-1)\Delta, i\Delta - 1]$, and it is inactive outside this interval, where $\Delta \geqslant 1$ is an unbounded parameter of the network. A possible specification for this network might require that each firing sequence is of the form $\mathbf{init}\ \mathbf{a}^*_{1,\ldots,N}\ \mathbf{a}_2\ \mathbf{a}^*_{2,\ldots,N} \ldots \mathbf{a}_i\ \mathbf{a}_i^*$ for some $1 \leqslant i \leqslant N$, and that v is increased only on the first occurrence of the events $\mathbf{a}_2, \ldots, \mathbf{a}_i$, in this order. This condition is encoded by the automaton B (Fig. 1, right). Observe that only the v variable is shared between the network $\langle A_1, \ldots, A_N \rangle$ and the specification

⋆ Supported by the French National Research Agency project VECOLIB (ANR-14-CE28-0018).
⋆⋆ Supported by the Czech Science Foundation project 14-11384S, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and the internal BUT project FIT-S-14-2486.

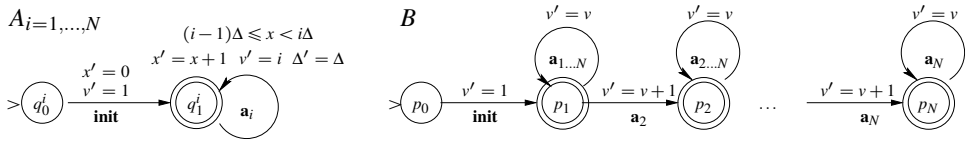


Fig. 1. An instance of the trace inclusion problem.

automaton B—we say that v is *observable* in this case. An example of a trace, for $\Delta = 2$ and $N \geqslant 3$, is: $(v=0)$ **init** $(v=1)\,\mathbf{a}_1\,(v=1)\,\mathbf{a}_1\,(v=1)\,\mathbf{a}_2\,(v=2)\,\mathbf{a}_2\,(v=2)\,\mathbf{a}_3\,(v=3)$. Our problem is to check that this, and all other traces of the network, are included in the language of the specification automaton, called the *observer*.

The trace inclusion problem has several applications, some of which we detail next. As the first potential application domain, we mention decision procedures for logics describing array structures in imperative programs [17, 16] that use a translation of array formulae to integer counter automata, which encode the set of array models of a formula. The expressiveness of such logics is currently limited by the decidability of the emptiness (reachability) problem for counter automata. If we give up on decidability, we can reduce an entailment between two array formulae to the trace inclusion of two integer counter automata, and use the method presented in this paper as a semi-decision procedure. To assess this claim, we have applied our trace inclusion method to several verification conditions for programs with unbounded arrays of integers [7].

Another application is within the theory of timed automata and regular specifications of timed languages [2] that can be both represented by finite automata extended with real-valued variables [14]. The verification problem boils down to the trace inclusion of two real-valued data automata. Our method has been tested on several timed verification problems, including communication protocols and boolean circuits [27].

When developing a method for checking the inclusion between trace languages of automata extended with variables ranging over infinite data domains, the first problem is the lack of determinization and/or complementation results. In fact, certain classes of infinite state systems, such as timed automata [2], cannot be determinized and are provably not closed under complement. This is the case due to the fact that the clock variables of a timed automaton are not observable in its timed language, which records only the time lapses between successive events. However, if we require that the values of all variables of a data automaton be part of its trace language, we obtain a determinization result, which generalizes the classical subset construction by taking into account the data valuations. Building on this first result, we define the complement of a data language and reduce the trace inclusion problem to the emptiness of a product data automaton $\mathcal{L}(A \times \overline{B}) = \varnothing$. It is crucial, for this reduction, that the variables $\mathbf{x}_B$ of the right-hand side data automaton B (the one being determinized) are also controlled by the left-hand side automaton A, in other words, that B has no hidden variables.

The language emptiness problem for data automata is, in general, undecidable [23]. Nevertheless, several semi-algorithms and tools for this problem (better known as the *reachability* problem) have been developed [3, 19, 22, 15]. Among those, the technique of *lazy predicate abstraction* [19] combined with *counterexample-driven refinement* us-

ing *interpolants* [22] has been shown to be particularly successful in proving emptiness of rather large infinite-state systems. Moreover, this technique shares similar aspects with the antichain-based algorithm for language inclusion in the case of a finite alphabet [1]. An important similarity is that both techniques use a partial order on states, to prune the state space during the search.

The main result of this paper is a semi-algorithm that combines the principle of the antichain-based language inclusion algorithm [1] with the interpolant-based abstraction refinement semi-algorithm [22], via a general notion of language-based subsumption relation. We have implemented our semi-algorithm in a prototype tool and carried out a number of experiments, involving hardware, real-time systems, and array logic problems. Since our procedure tests inclusion within a set of good traces, instead of empty intersection with a set of error traces, we can encode rather complex verification conditions concisely, by avoiding the blowup caused by an a-priori complementation of the automaton encoding the property.

1.1 Overview

We introduce the reader to our trace inclusion method by means of an example. For space reasons, all proofs are given in an extended version of the paper [21].

Let us consider the network of data automata $\langle A_1, A_2 \rangle$ and the data automaton B from Fig. 1. We prove that, for any value of Δ, any trace of the network $\langle A_1, A_2 \rangle$, obtained as an interleaving of the actions of A_1 and A_2, is also a trace of the observer B. To this end, our procedure will fire increasingly longer sequences of input events, in search for a counterexample trace. We keep a set of predicates associated with each state $(\langle q_1, q_2 \rangle, P)$ of the product automaton where q_i is a state of A_i and P is a set of states of B. These predicates are formulae that define over-approximations of the data values reached simultaneously by the network, when A_i is the state q_i, and by the observer B, in every state from P.

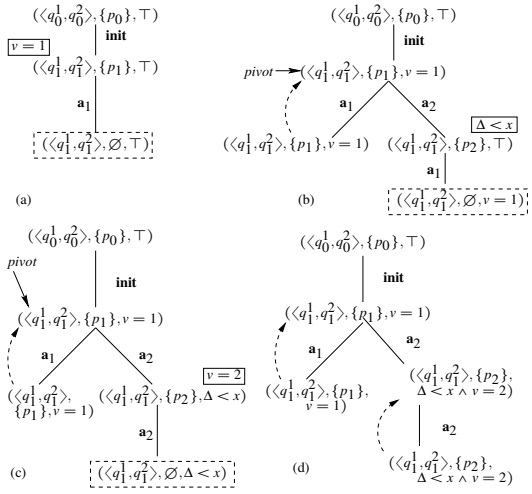


Fig. 2. Sample run of our semi-algorithm.

The first input event is **init**, on which A_1 and A_2 synchronize, moving together from the initial state $\langle q_0^1, q_0^2 \rangle$ to $\langle q_1^1, q_1^2 \rangle$. In response, B can chose to either (i) move from $\{p_0\}$ to $\{p_1\}$, matching the only transition rule from p_0, or (ii) ignore the transition rule and move to the empty set. In the first case, the values of v match the relation of the rule $p_0 \xrightarrow{\mathbf{init}, v'=1} p_1$, while in the second case, these values match the negated relation $\neg(v' = 1)$. The second case is impossible because the action of the network requires $x' = 0 \wedge v' = 1$. The only successor state is thus $(\langle q_1^1, q_1^2 \rangle, \{p_1\})$ in Fig. 2 (a). Since no predicates are initially available at this state, the best over-approximation of the set of reachable data valuations is the universal set ($\top$).

The second input event is $\mathbf{a}_1$, on which A_1 moves from q_1^1 back to itself, while A_2 makes an idle step because no transition with $\mathbf{a}_1$ is enabled from q_1^2. Again, B has the choice between moving from $\{p_1\}$ either to $\varnothing$ or $\{p_1\}$. Let us consider the first case, in which the successor state is $(\langle q_1^1, q_1^2\rangle, \varnothing, \top)$. Since q_1^1 and q_1^2 are final states of A_1 and A_2, respectively, and no final state of B is present in $\varnothing$, we say that the state is accepting. If the accepting state (in dashed boxes in Fig. 2) is reachable according to the transition constraints along the input sequence $\mathbf{init}.\mathbf{a}_1$, we have found a counterexample trace that is in the language of $\langle A_1, A_2\rangle$ but not in the language of B.

To verify the reachability of the accepting state, we check the satisfiability of the path formula corresponding to the composition of the transition constraints $x' = 0 \wedge v' = 1$ (**init**) and $0 \leqslant x < \Delta \wedge x' = x + 1 \wedge v' = 1 \wedge \neg(v' = v)$ ($\mathbf{a}_1$) in Fig. 2 (a). This formula is unsatisfiable, and the proof of infeasibility provides the interpolant $\langle v = 1\rangle$. This formula is an explanation for the infeasibility of the path because it is implied by the first constraint and it is unsatisfiable in conjunction with the second constraint. By associating the new predicate $v = 1$ with the state $(\langle q_1^1, q_1^2\rangle, \{p_1\})$, we ensure that the same spurious path will never be explored again.

We delete the spurious counterexample and recompute the states along the input sequence $\mathbf{init}.\mathbf{a}_1$ with the new predicate. In this case, $(\langle q_1^1, q_1^2\rangle, \varnothing)$ is unreachable, and the outcome is $(\langle q_1^1, q_1^2\rangle, \{p_1\}, v = 1)$. However, this state was first encountered after the sequence **init**, so there is no need to store a second occurrence of this state in the tree. We say that $\mathbf{init}.\mathbf{a}_1$ is subsumed by **init**, depicted by a dashed arrow in Fig. 2 (b).

We continue with $\mathbf{a}_2$ from the state $(\langle q_1^1, q_1^2\rangle, \{p_1\}, v = 1)$. In this case, A_1 makes an idle step and A_2 moves from q_1^2 to itself. In response, B has the choice between moving from $\{p_1\}$ to either (i) $\{p_1\}$ with the constraint $v' = v$, (ii) $\{p_2\}$ with the constraint $v' = v + 1$, (iii) $\{p_1, p_2\}$ with the constraint $v' = v \wedge v' = v + 1 \rightarrow \bot$ (this possibility is discarded), (iv) $\varnothing$ for data values that satisfy $\neg(v' = v) \wedge \neg(v' = v + 1)$. The last case is also discarded because the value of v after **init** constrained to 1 and the A_2 imposes further the constraint $v' = 2$ and $v = 1 \wedge v' = 2 \wedge \neg(v' = v) \wedge \neg(v' = v + 1) \rightarrow \bot$. Hence, the only $\mathbf{a}_2$-successor of $(\langle q_1^1, q_1^2\rangle, \{p_1\}, v = 1)$ is $(\langle q_1^1, q_1^2\rangle, \{p_2\}, \top)$, in Fig. 2 (b).

By firing the event $\mathbf{a}_1$ from this state, we reach $(\langle q_1^1, q_1^2\rangle, \varnothing, v = 1)$, which is, again, accepting. We check whether the path $\mathbf{init}.\mathbf{a}_2.\mathbf{a}_1$ is feasible, which turns out not to be the case. For efficiency reasons, we find the shortest suffix of this path that can be proved infeasible. It turns out that the sequence $\mathbf{a}_2.\mathbf{a}_1$ is infeasible starting from the state $(\langle q_1^1, q_1^2\rangle, \{p_1\}, v = 1)$, which is called the *pivot*. This proof of infeasibility yields the interpolant $\langle v = 1, \Delta < x\rangle$, and a new predicate $\Delta < x$ is associated with $(\langle q_1^1, q_1^2\rangle, \{p_2\})$. The refinement phase rebuilds only the subtree rooted at the pivot state, in Fig. 2 (b).

The procedure then builds the tree on Fig. 2 (c) starting from the pivot node and finds the accepting state $(\langle q_1^1, q_1^2\rangle, \varnothing, \Delta < x)$ as the result of firing the sequence $\mathbf{init}.\mathbf{a}_2.\mathbf{a}_2$. This path is spurious, and the new predicate $v = 2$ is associated with the location $(\langle q_1^1, q_1^2\rangle, \{p_2\})$. The pivot node is the same as in Fig. 2 (b), and, by recomputing the subtree rooted at this node with the new predicates, we obtain the tree in Fig. 2 (d), in which all frontier nodes are subsumed by their predecessors. Thus, no new event needs to be fired, and the procedure can stop reporting that the trace inclusion holds.

Related Work The trace inclusion problem has been previously addressed in the context of timed automata [25]. Although the problem is undecidable in general, decidabil-

ity is recovered when the left-hand side automaton has at most one clock, or the only constant appearing in the clock constraints is zero. These are essentially the only known decidable cases of language inclusion for timed automata.

The study of *data automata* [5, 6] usually deals with decision problems in logics describing data languages for simple theories, typically infinite data domains with equality. Although our notions of data words and data languages are similar to the classical ones in the literature [5, 6], the data automata defined in this paper are different from [5], as well as [6]. The main difference consists in the fact that the existing notions of data automata are controlled by equivalence relations of finite index, whereas in our case, the transitions are defined by unrestricted formulae in the first-order theory of the data domain. Moreover, the emptiness problems [5, 6] are decidable, whereas we consider an undecidable model that subsumes the existing ones.

Data words are also studied in the context of *symbolic visibly pushdown automata* (SVPA) [11]. Language inclusion is decidable for SVPAs with transition guards from a decidable theory because SVPAs are closed under complement and the emptiness can be reduced to a finite number of queries expressible in the underlying theory of guards. Decidability comes here at the cost of reducing the expressiveness and forbidding comparisons between adjacent positions in the input (only comparisons between matching call/return positions of the input nested words are allowed).

Another related model is that of *predicate automata* [13], which recognize languages over integer data by labeling the words with conjunctions of uninterpreted predicates. The emptiness problem is undecidable for this model and becomes decidable when all predicates are monadic. Exploring further the connection between predicate automata and our definition of data automata could also provide interesting examples for our method, stemming from verification problems for parallel programs.

Finally, several works on model checking infinite-state systems against CTL [4] and CTL* [9] specifications are related to our problem as they check inclusion between the set of computation trees of an infinite-state system and the set of trees defined by a branching temporal logic specification. The verification of existential CTL formulae [4] is reduced to solving forall-exists quantified Horn clauses by applying counterexample guided refinement to discover witnesses for existentially quantified variables. The work [9] on CTL* verification of infinite systems is based on partial symbolic determinization, using prophecy variables to summarize the future program execution. For finite-state systems, automata are a strictly more expressive formalism than temporal logics[3]. Such a comparison is, however, non-trivial for infinite-state systems. Nevertheless, we found the data automata considered in this paper to be a natural tool for specifying verification conditions of array programs [17, 16, 7] and regular properties of timed languages [2].

2 Data Automata

Let $\mathbb{N}$ denote the set of non-negative integers including zero. For any $k, \ell \in \mathbb{N}$, $k \leqslant \ell$, we write $[k, \ell]$ for the set $\{k, k+1, \ldots, \ell\}$. We write $\bot$ and $\top$ for the boolean constants

[3] For (in)finite words, the class of LTL-definable languages coincides with the star-free languages, which are a strict subclass of (ω-)regular languages.

false and *true*, respectively. Given a possibly infinite data domain $\mathcal{D}$, we denote by $\mathrm{Th}(\mathcal{D}) = \langle \mathcal{D}, p_1, \ldots, p_n, f_1, \ldots, f_m \rangle$ the set of syntactically correct first-order formulae with predicate symbols $p_1, \ldots, p_n$ and function symbols $f_1, \ldots, f_m$. A variable x is said to be *free* in a formula ϕ, denoted as $\phi(x)$, iff it does not occur under the scope of a quantifier.

Let $\mathbf{x} = \{x_1, \ldots, x_n\}$ be a finite set of variables. A *valuation* $\nu : \mathbf{x} \to \mathcal{D}$ is an assignment of the variables in $\mathbf{x}$ with values from $\mathcal{D}$. We denote by $\mathcal{D}^{\mathbf{x}}$ the set of such valuations. For a formula $\phi(\mathbf{x})$, we denote by $\nu \models_{\mathrm{Th}(\mathcal{D})} \phi$ the fact that substituting each variable $x \in \mathbf{x}$ by $\nu(x)$ yields a valid formula in the theory $\mathrm{Th}(\mathcal{D})$. In this case, ν is said to be a *model* of ϕ. A formula is said to be *satisfiable* iff it has a model. For a formula $\phi(\mathbf{x}, \mathbf{x}')$ where $\mathbf{x}' = \{x' \mid x \in \mathbf{x}\}$ and two valuations $\nu, \nu' \in \mathcal{D}^{\mathbf{x}}$, we denote by $(\nu, \nu') \models_{\mathrm{Th}(\mathcal{D})} \phi$ the fact that the formula obtained from ϕ by substituting each x with $\nu(x)$ and each x' with $\nu'(x')$ is valid in $\mathrm{Th}(\mathcal{D})$.

Data Automata. *Data Automata* (DA) are extensions of non-deterministic finite automata with variables ranging over an infinite data domain $\mathcal{D}$, equipped with a first order theory $\mathrm{Th}(\mathcal{D})$. Formally, a DA is a tuple $A = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q, \iota, F, \Delta \rangle$, where:

- Σ is a finite alphabet of input events and $\diamond \in \Sigma$ is a special padding symbol,
- $\mathbf{x} = \{x_1, \ldots, x_n\}$ is a set of variables,
- Q is a finite set of *states*, $\iota \in Q$ is an *initial* state, $F \subseteq Q$ are *final* states, and
- Δ is a set of *rules* of the form $q \xrightarrow{\sigma, \phi(\mathbf{x}, \mathbf{x}')} q'$ where $\sigma \in \Sigma$ is an alphabet symbol and $\phi(\mathbf{x}, \mathbf{x}')$ is a formula in $\mathrm{Th}(\mathcal{D})$.

A *configuration* of A is a pair $(q, \nu) \in Q \times \mathcal{D}^{\mathbf{x}}$. We say that a configuration (q', ν') is a *successor* of (q, ν) if and only if there exists a rule $q \xrightarrow{\sigma, \phi} q' \in \Delta$ and $(\nu, \nu') \models_{\mathrm{Th}(\mathcal{D})} \phi$. We denote the successor relation by $(q, \nu) \xrightarrow{\sigma, \phi}_A (q', \nu')$, and we omit writing ϕ and A when no confusion may arise. We denote by $succ_A(q, \nu) = \{(q', \nu') \mid (q, \nu) \to_A (q', \nu')\}$ the set of successors of a configuration (q, ν).

A *trace* is a finite sequence $w = (\nu_0, \sigma_0), \ldots, (\nu_{n-1}, \sigma_{n-1}), (\nu_n, \diamond)$ of pairs (ν_i, σ_i) taken from the infinite alphabet $\mathcal{D}^{\mathbf{x}} \times \Sigma$. A *run* of A over the *trace* w is a sequence of configurations $\pi : (q_0, \nu_0) \xrightarrow{\sigma_0} (q_1, \nu_1) \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-1}} (q_n, \nu_n)$. We say that the run π is *accepting* if and only if $q_n \in F$, in which case A *accepts* w. The *language* of A, denoted $\mathcal{L}(A)$, is the set of traces accepted by A.

Data Automata Networks. A *data automata network* (DAN) is a non-empty tuple $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ of data automata $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$, $i \in [1, N]$ whose sets of states are pairwise disjoint. A DAN is a succinct representation of an exponentially larger DA $\mathcal{A}^e = \langle \mathcal{D}, \Sigma_{\mathcal{A}}, \mathbf{x}_{\mathcal{A}}, Q_{\mathcal{A}}, \iota_{\mathcal{A}}, F_{\mathcal{A}}, \Delta_{\mathcal{A}} \rangle$, called the *expansion* of $\mathcal{A}$, where:

- $\Sigma_{\mathcal{A}} = \Sigma_1 \cup \ldots \cup \Sigma_N$ and $\mathbf{x}_{\mathcal{A}} = \mathbf{x}_1 \cup \ldots \cup \mathbf{x}_N$,
- $Q_{\mathcal{A}} = Q_1 \times \ldots \times Q_N$, $\iota_{\mathcal{A}} = \langle \iota_1, \ldots, \iota_N \rangle$ and $F_{\mathcal{A}} = F_1 \times \ldots \times F_N$,
- $\langle q_1, \ldots, q_N \rangle \xrightarrow{\sigma, \varphi} \langle q'_1, \ldots, q'_N \rangle$ if and only if (i) for each $i \in I$, there exists $\varphi_i \in \mathrm{Th}(\mathcal{D})$ such that $q_i \xrightarrow{\sigma, \varphi_i} q'_i \in \Delta_i$, (ii) for all $i \notin I$, $q_i = q'_i$, and (iii) $\varphi \equiv \bigwedge_{i \in I} \varphi_i \wedge \bigwedge_{j \notin I} \tau_j$, where $I = \{i \in [1, N] \mid q_i \xrightarrow{\sigma, \varphi_i} q'_i \in \Delta_i\}$ is the set of DA that can move from q_i to q'_i while reading the input symbol σ, and $\tau_j \equiv \bigwedge_{x \in \mathbf{x}_j \setminus (\bigcup_{i \in I} \mathbf{x}_i)} x' = x$ propagates the values of the local variables in A_j that are not updated by $\{A_i\}_{i \in I}$.

Intuitively, all automata that can read an input symbol synchronize their actions on that symbol whereas the rest of the automata make an idle step and copy the values of their

local variables which are not updated by the active automata. The language of the DAN $\mathcal{A}$ is defined as the language of its expansion DA, i.e. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^e)$.

Trace Inclusion. Let $\mathcal{A}$ be a DAN and $\mathcal{A}^e = \langle \mathcal{D}, \Sigma, \mathbf{x}_\mathcal{A}, Q_\mathcal{A}, \iota_\mathcal{A}, F_\mathcal{A}, \Delta_\mathcal{A} \rangle$ be its expansion. For a set of variables $\mathbf{y} \subseteq \mathbf{x}_\mathcal{A}$, we denote by $\nu{\downarrow_\mathbf{y}}$ the restriction of a valuation $\nu \in \mathcal{D}^{\mathbf{x}_\mathcal{A}}$ to the variables in $\mathbf{y}$. For a trace $w = (\nu_0, \sigma_0), \ldots, (\nu_n, \diamond) \in (\mathcal{D}^{\mathbf{x}_\mathcal{A}} \times \Sigma_\mathcal{A})^*$, we denote by $w{\downarrow_\mathbf{y}}$ the trace $(\nu_0{\downarrow_\mathbf{y}}, \sigma_0), \ldots, (\nu_{n-1}{\downarrow_\mathbf{y}}, \sigma_{n-1}), (\nu_n{\downarrow_\mathbf{y}}, \diamond) \in (\mathcal{D}^\mathbf{y} \times \Sigma)^*$. We lift this notion to sets of words in the natural way, by defining $\mathcal{L}(\mathcal{A}){\downarrow_\mathbf{y}} = \{ w{\downarrow_\mathbf{y}} \mid w \in \mathcal{L}(\mathcal{A}) \}$.

We are now ready to define the trace inclusion problem on which we focus in this paper. Given a DAN $\mathcal{A}$ as before and a DA $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ such that $\mathbf{x}_B \subseteq \mathbf{x}_\mathcal{A}$, the *trace inclusion problem* asks whether $\mathcal{L}(\mathcal{A}){\downarrow_{\mathbf{x}_B}} \subseteq \mathcal{L}(B)$? The right-hand side DA B is called *observer*, and the variables in $\mathbf{x}_B$ are called *observable* variables.

2.1 Boolean Closure Properties of Data Automata

Let $A = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q, \iota, F, \Delta \rangle$ be a DA for the rest of this section. A is said to be *deterministic* if and only if, for each trace $w \in \mathcal{L}(A)$, A has at most one run over w. The first result of this section is that, interestingly, any DA can be determinized while preserving its language. The determinization procedure is a generalization of the classical subset construction for Rabin-Scott word automata on finite alphabets. The reason why determinization is possible for automata over an infinite data alphabet $\mathcal{D}^\mathbf{x} \times \Sigma$ is that the successive values taken by *each variable* $x \in \mathbf{x}$ are tracked by the language $\mathcal{L}(A) \subseteq (\mathcal{D}^\mathbf{x} \times \Sigma)^*$. This assumption is crucial: a typical example of automata over an infinite alphabet, that cannot be determinized, are timed automata [2], where only the elapsed time is reflected in the language, and not the values of the variables (clocks).

Formally, the *deterministic* DA accepting the language $\mathcal{L}(A)$ is defined as $A^d = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q^d, \iota^d, F^d, \Delta^d \rangle$, where $Q^d = 2^Q$, $\iota^d = \{\iota\}$, $F^d = \{P \subseteq Q \mid P \cap F \neq \varnothing\}$ and Δ^d is the set of rules $P \xrightarrow{\sigma,\theta} P'$ such that:

- for all $p' \in P'$ there exists $p \in P$ and a rule $p \xrightarrow{\sigma,\psi} p' \in \Delta$,
- $\theta(\mathbf{x}, \mathbf{x}') \equiv \bigwedge_{p' \in P'} \bigvee_{\substack{p \xrightarrow{\sigma,\psi} p' \in \Delta \\ p \in P}} \psi \wedge \bigwedge_{p' \in Q \setminus P'} \bigwedge_{\substack{p \xrightarrow{\sigma,\varphi} p' \in \Delta \\ p \in P}} \neg\varphi$.

The main difference with the classical subset construction for Rabin-Scott automata is that here we consider *all sets* P' of states that have a predecessor in P, not just the maximal such set. The reason is that a set P' is not automatically subsumed by the union of all such sets due to the data constraints on the variables $\mathbf{x}$. Observe, moreover, that A^d can be built for any theory $\mathrm{Th}(\mathcal{D})$ that is closed under conjunction and negation.

Lemma 1. *Given a DA $A = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q, \iota, F, \Delta \rangle$, (1) for any $w \in (\mathcal{D}^\mathbf{x} \times \Sigma)^*$ and $P \in Q^d$, A^d has exactly one run on w that starts in P, and (2) $\mathcal{L}(A) = \mathcal{L}(A^d)$.*

The construction of a deterministic DA recognizing the language of A is key to defining a DA that recognizes the complement of A. Let $\overline{A} = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q^d, \iota^d, Q^d \setminus F^d, \Delta^d \rangle$. In other words, $\overline{A}$ has the same structure as A^d, and the set of final states consists of those subsets that contain no final state, i.e. $\{P \subseteq Q \mid P \cap F = \varnothing\}$. Using Lemma 1, it is not difficult to show that $\mathcal{L}(\overline{A}) = (\mathcal{D}^\mathbf{x} \times \Sigma)^* \setminus \mathcal{L}(A)$.

Next, we show closure of DA under intersection. Let $B = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q', \iota', F', \Delta' \rangle$ be a DA and define $A \times B = \langle \mathcal{D}, \Sigma, \mathbf{x}, Q \times Q', (\iota, \iota'), F \times F', \Delta^\times \rangle$, where $(q, q') \xrightarrow{\sigma,\varphi} (p, p') \in$

$\Delta^\times$ if and only if $q \xrightarrow{\sigma,\phi} p \in \Delta$, $q' \xrightarrow{\sigma,\psi} p' \in \Delta'$ and $\varphi \equiv \phi \wedge \psi$. It is easy to show that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$. DA are also closed under union, since $\mathcal{L}(A) \cup \mathcal{L}(B) = \overline{\mathcal{L}(\overline{A} \times \overline{B})}$.

Let us turn now to the trace inclusion problem. The following lemma shows that this problem can be effectively reduced to an equivalent language emptiness problem. However, note that this reduction does not work when the trace inclusion problem is generalized by removing the condition $\mathbf{x}_B \subseteq \mathbf{x}_A$. In other words, if the observer uses local variables not shared with the network[4], i.e. $\mathbf{x}_B \backslash \mathbf{x}_A \neq \varnothing$, the generalized trace inclusion problem $\mathcal{L}(A)\downarrow_{\mathbf{x}_A \cap \mathbf{x}_B} \subseteq \mathcal{L}(B)\downarrow_{\mathbf{x}_A \cap \mathbf{x}_B}$ has a negative answer iff *there exists a trace* $w = (\nu_0, \sigma_0), \ldots, (\nu_n, \diamond) \in \mathcal{L}(A)$ such that, *for all valuations* $\mu_0, \ldots, \mu_n \in \mathcal{D}^{\mathbf{x}_B \backslash \mathbf{x}_A}$, we have $w' = (\nu_0 \downarrow_{\mathbf{x}_A \cap \mathbf{x}_B} \cup \mu_0, \sigma_0), \ldots, (\nu_n \downarrow_{\mathbf{x}_A \cap \mathbf{x}_B} \cup \mu_n, \diamond) \notin \mathcal{L}(B)$. This kind of quantifier alternation cannot be easily accommodated within the framework of language emptiness, in which only one type of (existential) quantifier occurs.

Lemma 2. *Given DA* $A = \langle \mathcal{D}, \Sigma, \mathbf{x}_A, Q_A, \iota_A, F_A, \Delta_A \rangle$ *and* $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ *such that* $\mathbf{x}_B \subseteq \mathbf{x}_A$. *Then* $\mathcal{L}(A)\downarrow_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$ *if and only if* $\mathcal{L}(A \times \overline{B}) = \varnothing$.

The trace inclusion problem is undecidable, which can be shown by reduction from the language emptiness problem for DA (take B such that $\mathcal{L}(B) = \varnothing$). However the above lemma shows that any semi-decision procedure for the language emptiness problem can also be used to deal with the trace inclusion problem.

3 Abstract, Check, and Refine for Trace Inclusion

This section describes our semi-algorithm for checking the trace inclusion between a given network $\mathcal{A}$ and an observer B. Let $\mathcal{A}^e$ denote the expansion of $\mathcal{A}$, defined in the previous. In the light of Lemma 2, the trace inclusion problem $\mathcal{L}(\mathcal{A})\downarrow_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$, where the set of observable variables $\mathbf{x}_B$ is included in the set of network variables, can be reduced to the language emptiness problem $\mathcal{L}(\mathcal{A}^e \times \overline{B}) = \varnothing$.

Although language emptiness is undecidable for data automata [23], several cost-effective semi-algorithms and tools [18, 22, 15, 3] have been developed, showing that it is possible, in many practical cases, to provide a yes/no answer to this problem. However, to apply one of the existing off-the-shelf tools to our problem, one needs to build the product automaton $\mathcal{A}^e \times \overline{B}$ prior to the analysis. Due to the inherent state explosion caused by the interleaving semantics of the network as well as by the complementation of the observer, such a solution would not be efficient in practice.

To avoid building the product automaton, our procedure builds *on-the-fly* an over-approximation of the (possibly infinite) set of reachable configurations of $\mathcal{A}^e \times \overline{B}$. This over-approximation is defined using the approach of *lazy predicate abstraction* [18], combined with *counterexample-driven abstraction refinement* using *interpolants* [22]. We store the explored abstract states in a structure called an *antichain tree*. In general, antichain-based algorithms [28] store only states which are incomparable w.r.t. a partial order called *subsumption*. Our method can be thus seen as an extension of the antichain-based language inclusion algorithm [1] to infinite-state systems by means of predicate

[4] For timed automata, this is the case since the only shared variable is the time, and the observer may have local clocks.

abstraction and interpolation-based refinement. Since the trace inclusion problem is undecidable in general, termination of our procedure is not guaranteed; in the following, we shall, however, call our procedure an algorithm for the sake of brevity.

3.1 Antichain Trees

We define antichain trees, which are the main data structure of the trace inclusion procedure. Let $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ be a network of automata where $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$, for all $i \in [1,N]$, and let $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ be an observer such that $\mathbf{x}_B \subseteq \bigcup_{i=1}^{N} \mathbf{x}_i$. We also denote by $\mathcal{A}^e = \langle \mathcal{D}, \Sigma_\mathcal{A}, \mathbf{x}_\mathcal{A}, Q_\mathcal{A}, \iota_\mathcal{A}, F_\mathcal{A}, \Delta_\mathcal{A} \rangle$ the expansion of the network $\mathcal{A}$ and by $\mathcal{A}^e \times \overline{B} = \langle \mathcal{D}, \Sigma_\mathcal{A}, \mathbf{x}_\mathcal{A}, Q^p, \iota^p, F^p, \Delta^p \rangle$ the product automaton used for checking language inclusion.

An *antichain tree* for the network $\mathcal{A}$ and the observer B is a tree whose nodes are labeled by *product states* (see Fig. 2 for examples). Intuitively, a product state is an over-approximation of the set of configurations of the product automaton $\mathcal{A}^e \times \overline{B}$ that share the same control state. Formally, a *product state for $\mathcal{A}$ and B* is a tuple $s = (\mathbf{q}, P, \Phi)$ where (i) $(\mathbf{q}, P)$ is a state of $\mathcal{A}^e \times \overline{B}$ with $\mathbf{q} = \langle q_1, \ldots, q_N \rangle$ being a state of the network expansion $\mathcal{A}^e$ and P being a set of states of the observer B, and (ii) $\Phi(\mathbf{x}_\mathcal{A}) \in \mathrm{Th}(\mathcal{D})$ is a formula which defines an over-approximation of the set of valuations of the variables $\mathbf{x}_\mathcal{A}$ that reach the state $(\mathbf{q}, P)$ in $\mathcal{A}^e \times \overline{B}$. A product state $s = (\mathbf{q}, P, \Phi)$ is a finite representation of a possibly infinite set of configurations of $\mathcal{A}^e \times \overline{B}$, denoted as $[\![s]\!] = \{(\mathbf{q}, P, \nu) \mid \nu \models_{\mathrm{Th}(\mathcal{D})} \Phi\}$.

To build an over-approximation of the set of reachable states of the product automaton, we need to compute, for a product state s, an over-approximation of the set of configurations that can be reached in one step from s. To this end, we define first a finite abstract domain of product states, based on the notion of *predicate map*. A predicate map is a partial function that associates sets of facts about the values of the variables used in the product automaton, called *predicates*, with components of a product state, called *substates*. The reason behind the distribution of predicates over substates is twofold. First, we would like the abstraction to be *local*, i.e. the predicates needed to define a certain subtree in the antichain must be associated with the labels of that subtree only. Second, once a predicate appears in the context of a substate, it should be subsequently reused whenever that same substate occurs as part of another product state.

Formally, a *substate* of a state $(\langle q_1, \ldots, q_N \rangle, P) \in Q^p$ of the product automaton $\mathcal{A}^e \times \overline{B}$ is a pair $(\langle q_{i_1}, \ldots, q_{i_k} \rangle, S)$ such that (i) $\langle q_{i_1}, \ldots, q_{i_k} \rangle$ is a subsequence of $\langle q_1, \ldots, q_N \rangle$, and (ii) $S \neq \varnothing$ only if $S \cap P \neq \varnothing$. We denote the substate relation by $(\langle q_{i_1}, \ldots, q_{i_k} \rangle, S) \lhd (\langle q_1, \ldots, q_N \rangle, P)$. The substate relation requires the automata $A_{i_1}, \ldots, A_{i_k}$ of the network $\mathcal{A}$ to be in the control states $q_{i_1}, \ldots, q_{i_k}$ simultaneously, and the observer B to be in at least some state of S provided $S \neq \varnothing$ (if $S = \varnothing$, the state of B is considered to be irrelevant). Let $\mathcal{S}_{\langle \mathcal{A}, B \rangle} = \{r \mid \exists q \in Q^p \,.\, r \lhd q\}$ be the set of substates of a state of $\mathcal{A}^e \times \overline{B}$.

A *predicate map* $\Pi : \mathcal{S}_{\langle \mathcal{A}, B \rangle} \to 2^{\mathrm{Th}(\mathcal{D})}$ associates each substate $(\mathbf{r}, S) \in Q_{i_1} \times \ldots \times Q_{i_k} \times 2^{Q_B}$ with a set of formulae $\pi(\mathbf{x})$ where (i) $\mathbf{x} = \mathbf{x}_{i_1} \cup \ldots \cup \mathbf{x}_{i_k} \cup \mathbf{x}_B$ if $S \neq \varnothing$, and (ii) $\mathbf{x} = \mathbf{x}_{i_1} \cup \ldots \cup \mathbf{x}_{i_k}$ if $S = \varnothing$. Notice that a predicate associated with a substate refers only to the local variables of those network components $A_{i_1}, \ldots, A_{i_k}$ and of the observer B that occur in the particular substate.

We are now ready to define the abstract semantics of the product automaton $\mathcal{A}^e \times \overline{B}$, induced by a given predicate map. For convenience, we define first a set $Post(s)$ of *concrete successors* of a product state $s = (\mathbf{q}, P, \Phi)$ such that $(\mathbf{r}, S, \Psi) \in Post(s)$ if and only if (i) the product automaton $\mathcal{A}^e \times \overline{B}$ has a rule $(\mathbf{q}, P) \xrightarrow{\sigma,\theta} (\mathbf{r}, S) \in \Delta^p$ and $\Psi(\mathbf{x}_\mathcal{A}) \nrightarrow \bot$, where $\Psi(\mathbf{x}_\mathcal{A}) \equiv \exists \mathbf{x}'_\mathcal{A} \,.\, \Phi(\mathbf{x}'_\mathcal{A}) \wedge \theta(\mathbf{x}'_\mathcal{A}, \mathbf{x}_\mathcal{A})$. The set of concrete successors does not contain states with empty set of valuations; these states are unreachable in $\mathcal{A}^e \times \overline{B}$.

Given a predicate map Π, the set $Post_\Pi(s)$ of *abstract successors* of a product state s is defined as follows: $(\mathbf{r}, S, \Psi^\sharp) \in Post_\Pi(s)$ if and only if (i) there exists a product state $(\mathbf{r}, S, \Psi) \in Post(s)$ and (ii) $\Psi^\sharp(\mathbf{x}_\mathcal{A}) \equiv \bigwedge_{r \lhd (\mathbf{r}, S)} \bigwedge \{\pi \in \Pi(r) \mid \Psi \rightarrow \pi\}$. In other words, the set of data valuations that are reachable by an abstract successor is the tightest over-approximation of the concrete set of reachable valuations, obtained as the conjunction of the available predicates from the predicate map that over-approximate this set.

Finally, an *antichain tree* (or, simply antichain) $\mathcal{T}$ for $\mathcal{A}$ and B is a tree whose nodes are labeled with product states and whose edges are labeled by input symbols and concrete transition relations. Let $\mathbb{N}^*$ be the set of finite sequences of natural numbers that denote the positions in the tree. For a tree position $p \in \mathbb{N}^*$ and $i \in \mathbb{N}$, the position $p.i$ is a *child* of p. A set $S \subseteq \mathbb{N}^*$ is said to be *prefix-closed* if and only if, for each $p \in S$ and each prefix q of p, we have $q \in S$ as well. The root is denoted by the empty sequence ε.

Formally, an antichain $\mathcal{T}$ is a set of pairs $\langle s, p \rangle$, where s is a product state and $p \in \mathbb{N}^*$ is a tree position, such that (1) for each position $p \in \mathbb{N}^*$ there exists at most one product state s such that $\langle s, p \rangle \in \mathcal{T}$, (2) the set $\{p \mid \langle s, p \rangle \in \mathcal{T}\}$ is prefix-closed, (3) $(root_{\langle \mathcal{A}, B \rangle}, \varepsilon) \in \mathcal{T}$ where $root_{\langle \mathcal{A}, B \rangle} = (\langle \iota_1, \ldots, \iota_N \rangle, \{\iota_B\}, \top)$ is the label of the root, and (4) for each edge $(\langle s, p \rangle, \langle t, p.i \rangle)$ in $\mathcal{T}$, there exists a predicate map Π such that $t \in Post_\Pi(s)$. For the latter condition, if $s = (\mathbf{q}, P, \Phi)$ and $t = (\mathbf{r}, S, \Psi)$, there exists a unique rule $(\mathbf{q}, P) \xrightarrow{\sigma,\theta} (\mathbf{r}, S) \in \Delta^p$, and we shall sometimes denote the edge as $s \xrightarrow{\sigma,\theta} t$ or simply $s \xrightarrow{\theta} t$ when the tree positions are not important.

Each antichain node $n = (s, d_1 \ldots d_k) \in \mathcal{T}$ is naturally associated with a path from the root to itself $\rho : n_0 \xrightarrow{\sigma_1,\theta_1} n_1 \xrightarrow{\sigma_2,\theta_2} \ldots \xrightarrow{\sigma_2,\theta_k} n_k$. We denote by ρ_i the node n_i for each $i \in [0, k]$, and by $|\rho| = k$ the length of the path. The *path formula* associated with ρ is $\Theta(\rho) \equiv \bigwedge_{i=1}^{k} \theta(\mathbf{x}_\mathcal{A}^{i-1}, \mathbf{x}_\mathcal{A}^i)$ where $\mathbf{x}_\mathcal{A}^i = \{x^i \mid x \in \mathbf{x}_\mathcal{A}\}$ is a set of indexed variables.

3.2 Counterexample-driven Abstraction Refinement

A *counterexample* is a path from the root of the antichain to a node which is labeled by an *accepting* product state. A product state $(\mathbf{q}, P, \Phi)$ is said to be *accepting* iff $(\mathbf{q}, P)$ is an accepting state of the product automaton $\mathcal{A}^e \times \overline{B}$, i.e. $\mathbf{q} \in F_\mathcal{A}$ and $P \cap F_B = \varnothing$. A counterexample is said to be *spurious* if its path formula is unsatisfiable, i.e. the path does not correspond to a concrete execution of $\mathcal{A}^e \times \overline{B}$. In this case, we need to (i) remove the path ρ from the current antichain and (ii) refine the abstract domain in order to exclude the occurrence of ρ from future state space exploration.

Let $\rho : root_{\langle \mathcal{A}, B \rangle} = (\mathbf{q}_0, P_0, \Phi_0) \xrightarrow{\theta_1} (\mathbf{q}_1, P_1, \Phi_1) \xrightarrow{\theta_2} \ldots \xrightarrow{\theta_k} (\mathbf{q}_k, P_k, \Phi_k)$ be a spurious counterexample in the following. For efficiency reasons, we would like to save as much work as possible and remove only the smallest suffix of ρ which caused the spuriousness. For some $j \in [0, k]$, let $\Theta^j(\rho) \equiv \Phi_j(\mathbf{x}_\mathcal{A}^0) \wedge \bigwedge_{i=j}^{k} \theta_i(\mathbf{x}_\mathcal{A}^{i-j}, \mathbf{x}_\mathcal{A}^{i-j+1})$ be the formula

defining all sequences of data valuations that start in the set Φ_j and proceed along the suffix $(\mathbf{q}_j, P_j, \Phi_j) \rightarrow \ldots \rightarrow (\mathbf{q}_k, P_k, \Phi_k)$ of ρ. The *pivot* of a path ρ is the maximal position $j \in [0,k]$ such that $\Theta^j(\rho) = \bot$, and -1 if ρ is not spurious.

Finally, we describe the refinement of the predicate map, which ensures that a given spurious counterexample will never be found in a future iteration of the abstract state space exploration. The refinement is based on the notion of *interpolant* [22].

Definition 1. *Given a formula* $\Phi(\mathbf{x})$ *and a sequence* $\langle \theta_1(\mathbf{x},\mathbf{x}'), \ldots, \theta_k(\mathbf{x},\mathbf{x}') \rangle$ *of formulae, an* interpolant *is a sequence of formulae* $\mathbf{I} = \langle I_0(\mathbf{x}), \ldots, I_k(\mathbf{x}) \rangle$ *where: (1)* $\Phi \rightarrow I_0$*, (2)* $I_k \rightarrow \bot$*, and (3)* $I_{i-1}(\mathbf{x}) \wedge \theta_i(\mathbf{x},\mathbf{x}') \rightarrow I_i(\mathbf{x}')$ *for all* $i \in [1,k]$.

Any given interpolant is a witness for the unsatisfiability of a (suffix) path formula $\Theta^j(\rho)$. Dually, if *Craig's Interpolation Lemma* [10] holds for the considered first-order data theory $\mathrm{Th}(\mathcal{D})$, any infeasible path formula is guaranteed to have an interpolant.

Given a spurious counterexample ρ with pivot $j \geqslant 0$, an interpolant $\mathbf{I} = \langle I_0, \ldots, I_{k-j} \rangle$ for the infeasible path formula $\Theta^j(\rho)$ can be used to refine the abstract domain by augmenting the predicate map Π. As an effect of this refinement, the antichain construction algorithm will avoid every path with the suffix $(\mathbf{q}_j, P_j, \Phi_j) \rightarrow \ldots \rightarrow (\mathbf{q}_k, P_k, \Phi_k)$ in a future iteration. If $I_i \Leftrightarrow C_i^1(\mathbf{y}_1) \wedge \ldots \wedge C_i^{m_i}(\mathbf{y}_{m_i})$ is a conjunctive normal form (CNF) of the i-th component of the interpolant, we consider the substate $(\mathbf{r}_i^\ell, S_i^\ell)$ for each $C_i^\ell(\mathbf{y}_\ell)$ where $l \in [1, m_i]$:

- $\mathbf{r}_i^\ell = \langle q_{i_1}, \ldots, q_{i_h} \rangle$ where $1 \leqslant i_1 < \ldots < i_h \leqslant N$ is the largest sequence of indices such that $\mathbf{x}_{i_g} \cap \mathbf{y}_\ell \neq \varnothing$ for each $g \in [1,h]$ and the set $\mathbf{x}_{i_g}$ of variables of the network component DA A_{i_g},
- $S_i^\ell = P_j$ if $\mathbf{x}_B \cap \mathbf{y}_\ell \neq \varnothing$, and $S_i^\ell = \varnothing$, otherwise.

A predicate map Π is said to be *compatible* with a spurious path $\rho : s_0 \xrightarrow{\theta_1} \ldots \xrightarrow{\theta_k} s_k$ with pivot $j \geqslant 0$ if $s_j = (\mathbf{q}_j, P_j, \Phi_j)$ and there is an interpolant $\mathbf{I} = \langle I_0, \ldots, I_{k-j} \rangle$ of the suffix $\langle \theta_1, \ldots, \theta_k \rangle$ wrt. Φ_j such that, for each clause C of some equivalent CNF of I_i, $i \in [0, k-j]$, it holds that $C \in \Pi(r)$ for some substate $r \lhd s_{i+j}$. The following lemma proves that, under a predicate map compatible with a spurious path ρ, the antichain construction will exclude further paths that share the suffix of ρ starting with its pivot.

Lemma 3. *Let* $\rho : (\mathbf{q}_0, P_0, \Phi_0) \xrightarrow{\theta_0} (\mathbf{q}_1, P_1, \Phi_1) \xrightarrow{\theta_1} \ldots \xrightarrow{\theta_{k-1}} (\mathbf{q}_k, P_k, \Phi_k)$ *be a spurious counterexample and* Π *be a predicate map compatible with* ρ*. Then, there is no sequence of product states* $(\mathbf{q}_j, P_j, \Psi_0), \ldots, (\mathbf{q}_k, P_k, \Psi_{k-j})$ *such that: (1)* $\Psi_0 \rightarrow \Phi_j$ *and (2)* $(\mathbf{q}_{i+1}, P_{i+1}, \Psi_{i-j+1}) \in \mathit{Post}_\Pi((\mathbf{q}_i, P_i, \Psi_{i-j}))$ *for all* $i \in [j, k-1]$.

Observe that the refinement induced by interpolation is *local* since Π associates sets of predicates with substates of the states in $\mathcal{A}^e \times \overline{B}$, and the update impacts only the states occurring within the suffix of that particular spurious counterexample.

3.3 Subsumption

The main optimization of antichain-based algorithms [1] for checking language inclusion of automata over finite alphabets is that product states that are *subsets* of already

visited states are never stored in the antichain. On the other hand, language emptiness semi-algorithms, based on *predicate abstraction* [22] use a similar notion to cover newly generated abstract successor states by those that were visited sooner and that represent larger sets of configurations. In this case, state coverage does not only increase efficiency but also ensures termination of the semi-algorithm in many practical cases.

In this section, we generalize the subset relation used in classical antichain algorithms with the notion of coverage from predicate abstraction, and we define a more general notion of *subsumption* for data automata. Given a state $(\mathbf{q}, P)$ of the product automaton $\mathcal{A}^e \times \overline{B}$ and a valuation $\nu \in \mathcal{D}^{\mathbf{x}_\mathcal{A}}$, the *residual language* $\mathcal{L}_{(\mathbf{q},P,\nu)}(\mathcal{A}^e \times \overline{B})$ is the set of traces w accepted by $\mathcal{A}^e \times \overline{B}$ from the state $(\mathbf{q}, P)$ such that ν is the first valuation which occurs on w. This notion is then lifted to product states as follows: $\mathcal{L}_s(\mathcal{A}^e \times \overline{B}) = \bigcup_{(\mathbf{q},P,\nu)\in[\![s]\!]} \mathcal{L}_{(\mathbf{q},P,\nu)}(\mathcal{A}^e \times \overline{B})$ where $[\![s]\!]$ is the set of configurations of the product automaton $\mathcal{A}^e \times \overline{B}$ represented by the given product state s.

Definition 2. *Given a DAN $\mathcal{A}$ and a DA B, a partial order $\sqsubseteq$ is a* subsumption *provided that, for any two product states s and t, we have $s \sqsubseteq t$ only if $\mathcal{L}_s(\mathcal{A}^e \times \overline{B}) \subseteq \mathcal{L}_t(\mathcal{A}^e \times \overline{B})$.*

A procedure for checking the emptiness of $\mathcal{A}^e \times \overline{B}$ needs not continue the search from a product state s if it has already visited a product state t that subsumes s. The intuition is that any counterexample discovered from s can also be discovered from t. The trace inclusion semi-algorithm described below in Section 3.4 works, in principle, with any given subsumption relation. In practice, our implementation uses the subsumption relation defined by the lemma below:

Lemma 4. *The relation defined s.t. $(\mathbf{q}, P, \Phi) \sqsubseteq_{img} (\mathbf{r}, S, \Psi) \Leftrightarrow \mathbf{q} = \mathbf{r}$, $P \supseteq S$, and $\Phi \rightarrow \Psi$ is a subsumption.*

3.4 The Trace Inclusion Semi-algorithm

With the previous definitions, Algorithm 1 describes the procedure for checking trace inclusion. It uses a classical work-list iteration loop (lines 2-30) that builds an antichain tree by simultaneously unfolding the expansion $\mathcal{A}^e$ of the network $\mathcal{A}$ and the complement $\overline{B}$ of the observer B, while searching for a counterexample trace $w \in \mathcal{L}(\mathcal{A}^e \times \overline{B})$. Both $\mathcal{A}^e$ and $\overline{B}$ are built on-the-fly, during the abstract state space exploration.

The processed antichain nodes are kept in the set `Visited`, and their abstract successors, not yet processed, are kept in the set `Next`. Initially, $\texttt{Visited} = \varnothing$ and $\texttt{Next} = \{root_{\mathcal{A},B}\}$. The algorithm uses a predicate map Π, which is initially empty (line 1). We keep a set of subsumption edges $\texttt{Subsume} \subseteq \texttt{Visited} \times (\texttt{Visited} \cup \texttt{Next})$ with the following meaning: $(\langle s, p\rangle, \langle t, q\rangle) \in \texttt{Subsume}$ for two antichain nodes, where s, t are product states and $p, q \in \mathbb{N}^*$ are tree positions, if and only if there exists an abstract successor $s' \in Post_\Pi(s)$ such that $s' \sqsubseteq t$ (Definition 2). Observe that we do not explicitly store a subsumed successor of a product state s from the antichain; instead, we add a subsumption edge between the node labeled with s and the node that subsumes that particular successor. The algorithm terminates when each abstract successors of a node from `Next` is subsumed by some node from `Visited`.

An iteration of Algorithm 1 starts by choosing an antichain node $\texttt{curr} = \langle s, p\rangle$ from `Next` and moving it to `Visited` (line 3). If the product state s is accepting (line 5) we

Algorithm 1 Trace Inclusion Semi-algorithm

input:
1. a DAN $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ such that $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$ for all $i \in [1,N]$,
2. a DA $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ such that $\mathbf{x}_B \subseteq \bigcup_{i=1}^{N} \mathbf{x}_i$.

output: true if $\mathcal{L}(\mathcal{A})\downarrow_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$, otherwise a trace $\tau \in \mathcal{L}(\mathcal{A})\downarrow_{\mathbf{x}_B} \setminus \mathcal{L}(B)$.

1: $\Pi \leftarrow \emptyset$, `Visited` $\leftarrow \emptyset$, `Next` $\leftarrow \langle root_{\langle \mathcal{A},B \rangle}, \varepsilon \rangle$, `Subsume` $\leftarrow \emptyset$
2: **while** `Next` $\neq \emptyset$ **do**
3: choose `curr` $\in$ `Next` and move `curr` from `Next` to `Visited`
4: match `curr` with $\langle s, p \rangle$
5: **if** s is an accepting product state **then**
6: let ρ be the path from the root to `curr` and k be the pivot of ρ
7: **if** $k \geqslant 0$ **then**
8: $\Pi \leftarrow$ REFINEPREDICATEMAPBYINTERPOLATION(Π, ρ, k)
9: `rem` $\leftarrow$ SUBTREE(ρ_k)
10: **for** $(n,m) \in$ `Subsume` such that $m \in$ `rem` **do**
11: move n from `Visited` to `Next`
12: remove `rem` from (`Visited`, `Next`, `Subsume`)
13: add ρ_k to `Next`
14: **else**
15: return EXTRACTCOUNTEREXAMPLE(ρ)
16: **else**
17: $i \leftarrow 0$
18: **for** $t \in Post_{\Pi}(s)$ **do**
19: **if** there exists $m = \langle t', p' \rangle \in$ `Visited` such that $t \sqsubseteq t'$ **then**
20: add (`curr`, m) to `Subsume`
21: **else**
22: `rem` $\leftarrow \{n \in$ `Next` $\mid n = \langle t', p' \rangle$ and $t' \sqsubset t\}$
23: `succ` $\leftarrow \langle t, p.i \rangle$
24: $i \leftarrow i + 1$
25: **for** $n \in$ `Visited` such that n has a successor $m \in$ `rem` **do**
26: add $(n,$ `succ`$)$ to `Subsume`
27: **for** $(n,m) \in$ `Subsume` such that $m \in$ `rem` **do**
28: add $(n,$ `succ`$)$ to `Subsume`
29: remove `rem` from (`Visited`, `Next`, `Subsume`)
30: add `succ` to `Next`

check the counterexample path ρ, from the root of the antichain to `curr`, for spuriousness, by computing its pivot k. If $k \geqslant 0$, then ρ is a spurious counterexample (line 7), and the path formula of the suffix of ρ, which starts with position k, is infeasible. In this case, we compute an interpolant for the suffix and refine the current predicate map Π by adding the predicates from the interpolant to the corresponding substates of the product states from the suffix (line 8).

The computation of the interpolant and the update of the predicate map are done by the REFINEPREDICATEMAPBYINTERPOLATION function using the approach described in Section 3.2. Subsequently, we remove (line 12) from the current antichain the subtree rooted at the pivot node ρ_k, i.e. the k-th node on the path ρ (line 9), and add ρ_k to `Next` in order to trigger a recomputation of this subtree with the new predicate map. Moreover, all nodes with a successor previously subsumed by a node in the removed subtree are moved from `Visited` back to `Next` in order to reprocess them (line 11).

On the other hand, if the counterexample ρ is found to be real ($k = -1$), any valuation $\nu \in \bigcup_{i=0}^{|\rho|} \mathcal{D}^{\mathbf{x}^i_{\mathcal{A}}}$ that satisfies the path formula $\Theta(\rho)$ yields a counterexample trace $w \in \mathcal{L}(\mathcal{A})\downarrow_{\mathbf{x}_B} \setminus \mathcal{L}(B)$, obtained by ignoring all variables from $\mathbf{x}_{\mathcal{A}} \setminus \mathbf{x}_B$ (line 15).

If the current node is not accepting, we generate its abstract successors (line 18). In order to keep in the antichain only nodes that are incomparable w.r.t. the subsumption

relation $\sqsubseteq$, we add a successor t of s to `Next` (lines 23 and 30) only if it is not subsumed by another product state from a node $m \in$ `Visited`. Otherwise, we add a subsumption edge $(\texttt{curr}, m)$ to the set `Subsume` (line 20). Furthermore, if t is not subsumed by another state in `Visited`, we remove from `Next` all nodes $\langle t', p' \rangle$ such that t strictly subsumes t' (lines 22 and 29) and add subsumption edges to the node storing t from all nodes with a removed successor (line 26) or a removed subsumption edge (line 28).

The following theorem shows completeness modulo termination.

Theorem 1. *Let $\mathcal{A} = \langle A_1, \ldots, A_N \rangle$ be a DAN such that $A_i = \langle \mathcal{D}, \Sigma_i, \mathbf{x}_i, Q_i, \iota_i, F_i, \Delta_i \rangle$ for all $i \in [1, N]$, and let $B = \langle \mathcal{D}, \Sigma, \mathbf{x}_B, Q_B, \iota_B, F_B, \Delta_B \rangle$ be a DA such that $\mathbf{x}_B \subseteq \bigcup_{i=1}^{N} \mathbf{x}_i$. If Algorithm 1 terminates and returns true on input $\mathcal{A}$ and B, then $\mathcal{L}(\mathcal{A})\downarrow_{\mathbf{x}_B} \subseteq \mathcal{L}(B)$.*

The soundness question "if there exists a counterexample trace $w \in \mathcal{L}(\mathcal{A})\downarrow_{\mathbf{x}_B} \setminus \mathcal{L}(B)$, will Algorithm 1 discover it?" has a positive answer, when exploring paths in breadth-first order[5]. The reason is that any real counterexample corresponds to a finite path in the antichain, which will be eventually processed. Moreover, a real counterexample always results in an abstract counterexample, for any given predicate map.

4 Experimental Results

We have implemented Algorithm 1 in a prototype tool[6] using the MATHSAT SMT solver [8] for answering the satisfiability queries and computing the interpolants. The results of the experiments are given in Tables 1 and 2. The results were obtained on an Intel i7-4770 CPU @ 3.40GHz machine with 32GB RAM.

Table 1 contains experiments where the network $\mathcal{A}$ consists of a single component. We applied the tool on several verification conditions generated from imperative programs with arrays [7] (Array shift, Array rotation 1+2, Array split) available online [24]. Then, we applied it on models of hardware circuits (HW Counter 1+2, Synchronous LIFO) [26]. Finally, we checked two versions (correct and faulty) of the timed Alternating Bit Protocol [29].

Table 1. Experiments with single-component networks.

Example	A ($\lvert Q\rvert/\lvert\Delta\rvert$)	B ($\lvert Q\rvert/\lvert\Delta\rvert$)	Vars.	Res.	Time
Arrays shift	3/3	3/4	5	ok	$< 0.1s$
Array rotation 1	4/5	4/5	7	ok	$0.1s$
Array rotation 2	8/21	6/24	11	ok	$34s$
Array split	20/103	6/26	14	ok	$4m32s$
HW counter 1	2/3	1/2	2	ok	$0.2s$
HW counter 2	6/12	1/2	2	ok	$0.4s$
Synchr. LIFO	4/34	2/15	4	ok	$2.5s$
ABP-error	14/20	2/6	14	cex	$2s$
ABP-correct	14/20	2/6	14	ok	$3s$

Table 2 provides a list of experiments where the network $\mathcal{A}$ has $N > 1$ components. First, we have the example of Fig. 1 (Running). Next, we have several examples of real-time verification problems [27]: a controller of a railroad crossing [20] (Train) with T trains, the Fischer Mutual Exclusion protocol with deadlines Δ and Γ (Fischer), and a hardware communication circuit with K stages, composed of timed NOR gates (Stari). Third, we have modelled a Producer-Consumer example [12] with a fixed buffer size B. Fourth, we have experimented with several models of parallel programs that manipulate arrays (Array init, Array copy, Array join) with window size Δ.

[5] In fact, our implementation uses a queue to represent the `Next` set.

[6] `http://www.fit.vutbr.cz/research/groups/verifit/tools/includer/`

Table 2. Experiments with multiple-component networks (e.g., $2 \times 2/2 + 2 \times 3/3$ in column $\mathcal{A}$ means that $\mathcal{A}$ is a network with 4 components, of which 2 DA with 2 states and 2 rules, and 2 DA with 3 states and 3 rules).

Example	N	$\mathcal{A}$ ($\|Q\|/\|\Delta\|$)	B ($\|Q\|/\|\Delta\|$)	Vars.	Res.	Time
Running	2	2×2/2	3/4	3	ok	0.2*s*
Running	10	10×2/2	11/20	3	ok	25*s*
Train ($T = 5$)	7	5×3/3 + 4/4 + 4/4	2/38	1	ok	4*s*
Train ($T = 20$)	22	20×3/3 + 4/4 + 4/4	2/128	1	ok	6*m*26*s*
Fischer ($\Delta = 1, \Gamma = 2$)	2	2×5/6	1/10	4	ok	8*s*
Fischer ($\Delta = 1, \Gamma = 2$)	3	3×5/6	1/15	4	ok	2*m*48*s*
Fischer ($\Delta = 2, \Gamma = 1$)	2	2×5/6	1/10	4	cex	3*s*
Fischer ($\Delta = 2, \Gamma = 1$)	3	3×5/6	1/15	4	cex	32*s*
Stari ($K = 1$)	5	4/5 + 2/4 + 5/7 + 5/7 + 5/7	3/6	3	ok	0.5*s*
Stari ($K = 2$)	8	4/5 + 2/4 + 2×5/7 + 2×5/7 + 2×5/7	3/6	3	ok	0.5*s*
Prod-Cons ($B = 3$)	2	4/4 + 4/4	2/7	2	ok	10*s*
Prod-Cons ($B = 6$)	2	4/4 + 4/4	2/7	2	ok	2*m*32*s*
Array init ($\Delta = 2$)	5	5×2/2	2/6	2	ok	3*s*
Array init ($\Delta = 2$)	15	15×2/2	2/16	2	ok	3*m*15*s*
Array copy ($\Delta = 20$)	20	20×2/2	2/21	3	ok	0.3*s*
Array copy ($\Delta = 20$)	150	150×2/2	2/151	3	ok	43*s*
Array join ($\Delta = 10$)	4	2×2/2 + 2×3/3	2/3	2	ok	6*s*
Array join ($\Delta = 20$)	6	3×2/2 + 3×3/3	2/4	2	ok	1*m*9*s*

For the time being, our implementation is a proof-of-concept prototype that leaves plenty of room for optimization (e.g. caching intermediate computation results) likely to improve the performance on more complicated examples. Despite that, we found the results from Tables 1 and 2 rather encouraging.

5 Conclusions

We have presented an interpolation-based abstraction refinement method for trace inclusion between a network of data automata and an observer where the variables used by the observer are a subset of those used by the network. The procedure builds on a new determinization result for DAs and combines in a novel way predicate abstraction and interpolation with antichain-based inclusion checking. The procedure has been successfully applied to several examples, including verification problems for array programs, real-time systems, and hardware designs. Future work includes an extension of the method to data tree automata and its application to logics for heaps with data. Also, we foresee an extension of the method to handle infinite traces.

References

1. Abdulla, P., Chen, Y.F., Holik, L., Mayr, R., Vojnar, T.: When simulation meets antichains. In: Proc. of TACAS'10, LNCS, vol. 6015, pp. 158–174. Springer (2010)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
3. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: Fast acceleration of symbolic transition systems. In: Proc. of CAV'03. LNCS, vol. 2725. Springer (2003)

4. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: Proc. of CAV'13. LNCS, vol. 8044. Springer (2013)
5. Bojańczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data words. ACM Trans. Comput. Logic 12(4), 27:1–27:26 (2011)
6. Bouyer, P., Petit, A., Thrien, D.: An algebraic approach to data languages and timed languages. Information and Computation 182(2), 137 – 162 (2003)
7. Bozga, M., Habermehl, P., Iosif, R., Konecný, F., Vojnar, T.: Automatic verification of integer array programs. In: Proc. of CAV'09. LNCS, vol. 5643, pp. 157–172 (2009)
8. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Proc. of TACAS. LNCS, vol. 7795 (2013)
9. Cook, B., Khlaaf, H., Piterman, N.: On automation of ctl* verification for infinite-state systems. In: Proc. of CAV'15. LNCS, vol. 9206. Springer (2015)
10. Craig, W.: Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. J. Symb. Log. 22(3), 269–285 (1957)
11. D'Antoni, L., Alur, R.: Symbolic visibly pushdown automata. In: Proc. of CAV'14. LNCS, vol. 8559. Springer (2014)
12. Dhar, A.: Algorithms For Model-Checking Flat Counter Systems. Ph.D. thesis, Univ. Paris 7 (2014)
13. Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. SIGPLAN Not. 50(1), 407–420 (Jan 2015)
14. Fribourg, L.: A closed-form evaluation for extended timed automata. Tech. rep., CNRS et Ecole Normale Supérieure de Cachan (1998)
15. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 405–416 (2012)
16. Habermehl, P., Iosif, R., Vojnar, T.: A logic of singly indexed arrays. In: Proc. of LPAR'08. LNCS, vol. 5330, pp. 558–573 (2008)
17. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: Proc. of FOSSACS'08. LNCS, vol. 4962, pp. 474–489 (2008)
18. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. of POPL'02. ACM (2002)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with Blast. In: Proc. of 10th SPIN Workshop. LNCS, vol. 2648 (2003)
20. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation 111, 394–406 (1992)
21. Iosif, R., Rogalewicz, A., Vojnar, T.: Abstraction refinement for trace inclusion of data automata. CoRR abs/1410.5056 (2014), http://arxiv.org/abs/1410.5056
22. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. of CAV'06. LNCS, vol. 4144. Springer (2006)
23. Minsky, M.: Computation: Finite and Infinite Machines. Prentice-Hall (1967)
24. Numerical Transition Systems Repository. http://http://nts.imag.fr/index.php/Flata (2012)
25. Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: Closing a decidability gap. In: Proc of LICS'04. IEEE Computer Society (2004)
26. Smrcka, A., Vojnar, T.: Verifying parametrised hardware designs via counter automata. In: HVC'07. pp. 51–68 (2007)
27. Tripakis, S.: The analysis of timed systems in practice. Ph.D. thesis, Univ. Joseph Fourier, Grenoble (December 1998)
28. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.: Antichains: A new algorithm for checking universality of finite automata. In: Proc. of CAV'06. LNCS, vol. 4144. Springer (2006)
29. Zbrzezny, A., Polrola, A.: Sat-based reachability checking for timed automata with discrete data. Fundamenta Informaticae 79, 1–15 (2007)