Harnessing Forest Automata for Verification of Heap Manipulating Programs

Jiri Simacek

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

préparée dans le cadre d'une cotutelle entre l'*Université de Grenoble* et *Vysoké učení technické v Brně*

Spécialité : **Mathematiques, Sciences et Technologie de l'Information, Informatique**

Arrêté ministériel : le 6 janvier 2005 - 7 août 2006

Présentée par

« Jiří Šimáček »

Thèse dirigée par **« Tomáš Vojnar »** et **« Yassine Lakhnech »**
codirigée par **« Radu Iosif »**

préparée au sein de **Laboratoire VERIMAG**

dans **l'École Doctorale Mathematiques, Sciences et Technologie de l'Information, Informatique**

Vérification de programmes avec structures de données complexes

Thèse soutenue publiquement le **« 29/10/2012 »**,
devant le jury composé de :

professeur Parosh Abdulla
Rapporteur
professeur Mojmír Křetínský
Rapporteur
professeur Ahmed Bouajjani
Membre
professeur Petr Jančar
Membre
professeur agrégé David Monniaux
Membre
professeur Milan Češka
Président

Université Joseph Fourier / Université Pierre Mendès France /
Université Stendhal / Université de Savoie / Grenoble INP

Abstract

This work addresses verification of infinite-state systems, more specifically, verification of programs manipulating complex dynamic linked data structures. Many different approaches emerged to date, but none of them provides a sufficiently robust solution which would succeed in all possible scenarios appearing in practice. Therefore, in this work, we propose a new approach which aims at improving the current state of the art in several dimensions. Our approach is based on using tree automata, but it is also partially inspired by some ideas taken from the methods based on separation logic. Apart from that, we also present multiple advancements within the implementation of various tree automata operations, crucial for our verification method to succeed in practice. Namely, we provide an optimised algorithm for computing simulations over labelled transition systems which then translates into more efficient computation of simulations over tree automata. We also give a new algorithm for checking inclusion over tree automata, and we provide experimental evaluation demonstrating that the new algorithm outperforms other existing approaches.

Keywords

pointers, heaps, verification, shape analysis, regular model checking, finite tree automata, antichains, simulations, language inclusion

Abstrakt

Tato práce se zabývá verifikací nekonečně stavových systémů, konkrétně, verifikací programů využívajích složité dynamicky propojované datové struktury. V minulosti se k řešení tohoto problému objevilo mnoho různých přístupů, avšak žádný z nich doposud nebyl natolik robustní, aby fungoval ve všech případech, se kterými se lze v praxi setkat. Ve snaze poskytnout vyšší úroveň automatizace a současně umožnit verifikaci programů se složitějšími datovými strukturami v této práci navrhujeme nový přístup, který je založen zejména na použití stromových automatů, ale je také částečně inspirován některými myšlenkami, které jsou převzaty z metod založených na separační logice. Mimo to také představujeme několik vylepšení v oblasti implementace operací nad stromovými automaty, které jsou klíčové pro praktickou využitelnost navrhované verifikační metody. Konkrétně uvádíme optimalizovaný algoritmus pro výpočet simulací pro přechodový systém s návěštími, pomocí kterého lze efektivněji počítat simulace pro stromové automaty. Dále uvádíme nový algoritmus pro testování inkluze stromových automatů společně s experimenty, které ukazují, že tento algoritmus překonává jiné existující přístupy.

Résumé

Les travaux décrits dans cette thèse portent sur le problème de vérification des systèmes avec espaces d'états infinis, et, en particulier, avec des structures de données chaînées. Plusieurs approches ont émergé, sans donner des solutions convenables et robustes, qui pourrait faire face aux situations rencontrées dans la pratique. Nos travaux proposent une approche nouvelle, qui combine les avantages de deux approches très prometteuses: la représentation symbolique a base d'automates d'arbre, et la logique de séparation. On présente également plusieurs améliorations concernant l'implementation de différentes opérations sur les automates d'arbre, requises pour le succès pratique de notre méthode. En particulier, on propose un algorithme optimise pour le calcul des simulations sur les systèmes de transitions étiquettes, qui se traduit dans un algorithme efficace pour le calcul des simulations sur les automates d'arbre. En outre, on présente un nouvel algorithme pour le problème d'inclusion sur les automates d'arbre. Un nombre important d'expérimentes montre que cet algorithme est plus efficace que certaines des méthodes existantes.

Acknowledgements

I am deeply grateful to both of my supervisors Tomáš Vojnar and Radu Iosif for their great guidance and help without which a successful completion of this work would not be possible. I would also like to thank other coauthors, namely Peter Habermehl, Lukáš Holík, Ondřej Lengál, and Adam Rogalewicz for their great cooperation during our common research. Furthermore, I would like to thank Lukáš Holík and especially Tomáš Vojnar once more for providing valuable comments regarding this text. Also, I want to thank Kamil Dudka for creating a plugin infrastructure which simplified the development of the verification tool presented in this thesis. Finally, I am also grateful to my parents and the rest of my family for the moral support that they always provided.

The work presented in this thesis was supported by the Czech Ministry of Education (project COST OC10009, Czech-French Barrande project MEB021023, the long-term institutional project MSM0021630528), the Czech Science Foundation (projects P103/10/0306, 102/09/H042, 201/09/P531), the EU/Czech IT4Innovations Centre of Excellence (project ED1.1.00/02.0070), the European Science Foundation (ESF COST action IC0901), the French National Research Agency (project ANR-09-SEGI-016 VERIDYC), and the Brno University of Technology (projects FIT-S-10-1, FIT-S-11-1, FIT-S-12-1).

Contents

1 Introduction

Traditional approaches for ensuring quality of computer systems such as code review or testing are nowadays reaching their inherent limitations due to the growing complexity of the current computer systems. That is why, there is an increasing demand for more capable techniques. One of the ways how to deal with this situation is to use suitable formal verification approaches.

In case of software, one especially critical area is that of ensuring safe memory usage in programs using dynamic memory allocation. The development of such programs is quite complicated, and many programming errors can easily arise here. Worse yet, the bugs within memory manipulation often cause an unpredictable behaviour, and they are often very hard to find. Indeed, despite the use of testing and other traditional means of quality assurance, many of the memory errors make it into the production versions of programs causing them to crash unexpectedly by breaking memory protection or to gradually waste more and more memory (if the error causes memory leaks). Consequently, using formal verification is highly desirable in this area.

Formal verification of programs with dynamically linked data structures is, however, very demanding since these programs are infinite-state. One of the most promising ways of dealing with infinite state verification is to use symbolic verification in which infinite sets of reachable configurations are represented finitely using a suitable formalism. In case of programs with dynamically linked data structures, the use of symbolic verification is complicated by the fact that their configurations are graphs, and representing infinite sets of graphs is particularly complicated (compared to objects like words or trees).

Many different verification approaches for programs manipulating dynamically linked data structures have emerged so far. Some of them are based on logics [MS01, SRW02, Rey02, BCC+07, GVA07, NDQC07, CRN07, ZKR08, YLB+08, CDOY09, MPQ11, DPV11], others are based on using automata [BHRV06b, BBH+11, DEG06], upward closed sets [ABC+08, ACV11], as well as other formalisms. The approaches differ in their generality, efficiency, and degree of automation. Among the fully automatic ones, the works [BCC+07, YLB+08] present an approach based on separation logic (see [Rey02]) that is quite scalable due to using local reasoning. However, their method is limited to programs manipulating various kinds of lists. There are other works based on separation logic which also consider trees or even more complex data structures, but they either expect the input program to be in some special form (e.g., [GVA07]) or they require some additional information about the data structures which are involved (as in [NDQC07, MTLT10]). Similarly, even the other existing approaches that are not based on separation logic often suffer from the need of non-trivial user aid in order to successfully finish the verification task (see, e.g., [MS01, SRW02]). On the other hand, the work [BHRV06b] proposed an

automata-based method which is able to handle fully automatically quite complex data structures, but it suffers from several drawbacks such as a monolithic representation of memory configurations which does not allow this approach to scale well.

Another issue with many existing automata-based approaches for symbolic verification of infinite-state systems (such as programs with dynamically linked data structures) is that they are based on using *deterministic finite automata* (DFA). This allows them to take advantage of the relatively simple and well-established algorithms for computing standard operations such as language union, language inclusion, minimisation, complementation, etc. However, some of these operations internally produce nondeterministic finite automata which then need to be immediately determinised. This is not difficult in theory, but in practice, the size of the automata for which the operation can be computed is very limited as the size of the corresponding deterministic automata can be exponential in the size of the original nondeterministic ones. As a result, verification methods based on using DFA do not perform that well when they are forced to work with automata of bigger size.

A use of nondeterministic finite automata (NFA) was proposed in [BHH$^+$08] in an effort to address the issues of scalability of symbolic automata-based verification methods. Despite the fact that this approach cannot improve the theoretical worst-case complexity, it turns out that the use of nondeterministic automata can greatly improve the scalability of automata-based verification approaches in practice. However, in order to be able to efficiently use NFA in the given context, one needs to have available suitable algorithms for certain critical automata operations that will perform these operations without necessarily determinising the automata. This is in particular the case of language inclusion, minimisation (or, more precisely, size reduction), and complementation (if needed). Some of these algorithms have already been proposed (e.g., [DWDHR06, BHH$^+$08] use *antichains* to deal with the problem of language inclusion), but there remained a significant space for improvement. In particular, within the algorithms for language inclusion presented in [ACH$^+$10, DR10] (which further optimise the work of [DWDHR06, BHH$^+$08]) and size reduction presented in [ABH$^+$08], one has to compute the maximal simulation relation over the set of states of an automaton. It turns out that the computation of the simulation relation often takes the majority of the time, especially in the case of size reduction. Hence, efficient techniques for computing simulations are needed. Moreover, the technique of [ACH$^+$10] for antichain-based inclusion checking of TA uses upward simulations which are especially costly to compute and often very sparse. Hence, there is also a need of still better inclusion checking on NTA.

1.1 Goals of the Work

Above, we have argued that development of programs with dynamically linked data structures is difficult, error-prone, and the errors arising in this kind of programs are difficult to discover using traditional approaches for quality as-

surance. Hence, there is a strong need for formal verification approaches in this area. However, most of the existing formal verification techniques for programs with dynamically linked data structures are either not fully automatic, or they can handle only a limited class of data structures. On the other hand, those techniques that can automatically handle complex data structures are usually computationally very expensive. Therefore, our first goal is to develop an efficient and fully automatic approach for verification of this kind of programs. The new approach is intended to be able to verify programs manipulating more complex data structures than those that can be efficiently handled by existing fully automatic methods. We, in particular, focus on combining automata-based approaches (which are rather general and which come with flexible and refinable abstraction) with some principles taken from the quite scalable methods based on separation logic, which is especially the case of local reasoning.

The second goal is to further improve the available algorithms implementing the operations that one needs to perform over nondeterministic automata when using them in some method for symbolic verification of infinite-state systems such as the one proposed within the first goal. Concretely, our aim is to improve algorithms for inclusion checking by considering the so-far neglected top-down approach and to improve automata reduction by providing a better algorithm for computing simulations.

1.2 An Overview of the Achieved Results

In this section, we summarise the contributions that we have achieved within the particular areas marked out by the goals of the work.

Verification of Heap Manipulating Programs. We propose a novel method for symbolic verification of heap manipulating programs. The main idea of our approach is the following. We represent heap graphs via their canonical *tree decomposition*. This can be done thanks to the observation that every heap graph can be decomposed into a set of *tree components* when the leaves of the tree components are allowed to refer back to the roots of these components. Moreover, given a total ordering on program variables and pointer links (called selectors), each heap graph may be decomposed into a tuple of tree components in a *canonical way*. 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 the particular cut-points. These components should 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.

We introduce a new formalism of forest automata upon the described decomposition of heaps into tree components in order to be able to efficiently represent sets of such decompositions (and hence sets of heaps). In particular, a *forest automaton* (FA) is basically a tuple of tree automata. Each of the tree automata within the tuple accepts trees whose leaves may refer back to the roots of any of these trees. A forest automaton then represents exactly the set of heaps that may be obtained by taking a single tree from the language of each of the component tree automata and by gluing the roots of the trees with the leaves referring to them.

Further, we show that FA enjoy some nice properties, which are crucial for our verification approach. In particular, we show that relevant C statements can be easily symbolically executed over forest automata. Moreover, one can implement efficient abstraction on FA as well as decide language inclusion (which is needed for fixpoint checking) The latter can in particular be implemented by an easy reduction to the well-known problem of language inclusion of tree automata.

Next, in order to extend the class of graphs that can be handled in our framework, we extend FA to hierarchically nested FA by allowing their alphabet symbols to encode sets of subgraphs instead of plain hyperedges. These sets of subgraphs are again represented using hierarchically nested FA. For the hierarchical FA, we do not obtain the same nice theoretical properties, but we at least show that the needed operations (such as language inclusion checking) can be sufficiently precisely approximated (building on the results for plain FA).

In our symbolic verification approach, a symbolic state is thus composed of a finite number of program variable assignments, a forest automaton which is able to represent infinitely many heaps, and a program counter specifying which instruction of the verified code is to be executed in the next step. We have implemented the approach in a tool called Forester in order to experimentally evaluate our method. The results show that the tool is very competitive when compared to other existing tools for verification of dynamic data structures while being quite general and fully automatic.

Simulations over Labelled Transition Systems and Tree Automata. We address the problem of computing simulations over a labelled transition system (LTS) by designing an optimised version of the algorithm proposed in [ABH$^+$08, AHKV08] (which is itself based on the algorithms for Kripke structures from [HHK95, RT07]). Our optimisation is based on the observation that in practice, we often work with LTSs in which transitions leading from particular states are labelled by some subset of all alphabet symbols only. By a careful analysis of the original algorithm, we have identified that one can exploit this irregular use of alphabet symbols in order to improve the performance of the computation.

In particular, for two states p and q within an LTS, p can simulate q only if for any transition leading from q labelled by some symbol a, there is a transition leading from p labelled by a. Using this fact, we refine the initial estimation of the simulation relation within the first phase of the algorithm introduced in

[AHKV08], and we show that, thanks to the initial refinement, certain iterations of the algorithm can be skipped without affecting its output. Furthermore, we show that certain parts of the data structures used by the original algorithm are no longer needed when our optimisation is used. Hence, we obtain a reduction of the space requirements, too. For our optimised algorithm, we also derive its worst-case time and space complexity.

As shown in [AHKV08], simulations over tree automata can be efficiently computed via a translation into LTSs. Therefore, we also derive the complexity of computing simulations over tree automata when our optimised algorithm is applied on LTSs produced during the translation. In this case, we achieve a promising reduction of the asymptotic complexity. Moreover, we validate the theoretical results by an experimental evaluation demonstrating significant savings in terms of space as well as time on both LTSs and tree automata.

Language Inclusion Checking for Tree Automata. For the purposes of our verification technique for programs manipulating dynamically linked data structures, we also investigate new efficient methods for checking language inclusion on nondeterministic tree automata. Originally, we intended to build upon the bottom-up inclusion checking introduced in [ACH+10] which is based on combining antichains with upward simulation. However, during our experiments, we have realised that the particular combination does not yield the expected improvements in the efficiency of inclusion checking because the computation of upward simulation is often too costly. In reaction to this issue, we have designed a new top-down inclusion checking algorithm which is of a similar spirit as the one in [HVP05], but it is not limited to binary trees, and it is optimised in several crucial ways as described below.

Unlike the bottom-up approach which starts in the leaves and proceeds towards the roots, the top-down inclusion starts in roots (represented via accepting states) and continues towards the leaves. The approach is based on generating pairs in which the first component corresponds to a state of the first automaton, and the second component contains a set of states of the second automaton. During the computation, the algorithm maintains a set of those pairs for which the inclusion has been shown not to hold. A fundamental problem of this method is the fact that the number of successor pairs one needs to explore grows exponentially with the level of the (top-down) nondeterminism of tree automata. Due to this, the construction may blow up and run out of the available time on certain automata pairs. We, however, show that it is often possible to work around this issue by using the principle of antichains [DWDHR06]. Moreover, we further improve the approach by combining it with a use of downward simulation which greatly reduces the risk of the blow-up. Finally, we also present a sophisticated modification of the algorithm which allows us to remember and to exploit the pairs for which the inclusion holds (apart from those in which it does not).

We have implemented explicit and semi-symbolic variants of the various, above mentioned language inclusion algorithms. Our experiments with the bottom-up and the top-down approaches for checking language inclusion of

tree automata show that the top-down inclusion checking dominates in most of our benchmarks.

An Efficient Library for Dealing with NTA. The proposed algorithms for inclusion checking and simulation computation have been incorporated into a newly designed library (called VATA) for dealing with NTA, together with some further operations such as simulation-based reduction, union, intersection, etc. Various lower-level optimisations of the basic algorithms have been proposed within the implementation of the library to make it as efficient as possible. This, in particular, includes various improvements of the bottom up inclusion checking of [ACH+10] which make it more efficient in practice. Another significant improvement introduced in the implementation of VATA is a substantial refinement of the internal representation of the data structures used in the algorithm for computing simulations which further reduce its memory footprint.

1.3 Plan of the Thesis

Chapter 2 contains preliminaries on labelled transition systems, tree automata, and simulations. Chapter 3 proposes the notion of forest automata which serves as a theoretical basis for our verification technique for programs manipulating dynamically linked data structures. In Chapter 4, we provide a detailed description of the verification procedure as well as the experimental evaluation of our prototype tool Forester based on it. Our optimised algorithm for computing simulations on LTSs is presented in Chapter 5. Chapter 6 describes several variants of top-down inclusion checking algorithms. Essentials of our tree automata library based on the proposed algorithms are discussed in Chapter 7, including various lower-level optimisations of the implementation of the algorithms discussed in Chapter 5 and Chapter 6. Finally, Chapter 8 concludes the thesis.

2 Preliminaries

In this chapter, we introduce preliminaries on labelled transition systems, alphabets, trees, tree automata, and simulations that we build on in this work.

2.1 Labelled Transition Systems

A *labeled transition system (LTS)* is a tuple $T = (S, \Sigma, \{\delta_a \mid a \in \Sigma\})$, where S is a finite set of states, Σ is a finite set of labels, and for each $a \in \Sigma$, $\delta_a \subseteq S \times S$ is an a-labeled transition relation. We use δ to denote $\bigcup_{a \in \Sigma} \delta_a$.

2.2 Alphabets and Trees

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

2.3 Tree Automata

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

For an automaton $\mathcal{A} = (Q, \Sigma, \Delta, F)$, we use $Q^\#$ to denote the set of all tuples of states from Q with up to the maximum arity that some symbol in

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

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

2.4 Simulations over Tree Automata

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

Given two sets S, Q such that $S \subseteq Q$ and a preorder $\preceq \subseteq Q \times Q$, then $S^\preceq$ denotes the set $\{q \in S : \nexists q' \in S. q \preceq q'\}$.

2.5 Regular Tree Model Checking

(This section is borrowed from [Hol11] with the kind permission of the author.)

Regular tree model checking (RTMC) [Sha01, BT02, ALdR05, BHV04] is a general and uniform framework for verifying infinite-state systems. In RTMC, configurations of a system being verified are encoded by trees, sets of the configurations by tree automata, and transitions of the verified system by a term rewriting system (usually given as a tree transducer or a set of tree transducers). Then, verification problems based on performing reachability analysis correspond to computing closures of regular languages under rewriting systems, i.e., given a term rewriting system τ and a regular tree language I, one needs

to compute $\tau^*(I)$ where τ^* is the reflexive-transitive closure of τ. This computation is impossible in general. Therefore, the main issue in RTMC is to find accurate and powerful fixpoint acceleration techniques helping the convergence of computing language closures. One of the most successful acceleration techniques used in RTMC is abstraction whose use leads to the so-called *abstract regular tree model checking* (ARTMC) [BHRV06a, BHRV06b], on which we concentrate in this work.

Abstract Regular Tree Model Checking. We briefly recall the basic principles of ARTMC in the way they were introduced in [BHRV06b]. Let Σ be a ranked alphabet and $\mathbb{M}_\Sigma$ the set of all tree automata over Σ. Let $\mathcal{I} \in \mathbb{M}_\Sigma$ be a tree automaton describing a set of initial configurations, τ a term rewriting system describing the behaviour of a system, and $\mathcal{B} \in \mathbb{M}_\Sigma$ a tree automaton describing a set of bad configurations. The safety verification problem can now be formulated as checking whether the following holds:

$$\tau^*(\mathcal{L}(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) = \emptyset \tag{2.1}$$

In ARTMC, the precise set of reachable configurations $\tau^*(\mathcal{L}(\mathcal{I}))$ is not computed to solve Problem (2.1). Instead, its overapproximation is computed by interleaving the application of τ and the union in $\mathcal{L}(\mathcal{I}) \cup \tau(\mathcal{L}(\mathcal{I})) \cup \tau(\tau(\mathcal{L}(\mathcal{I}))) \cup \ldots$ with an application of an abstraction function α. The abstraction is applied on the tree automata encoding the so-far computed sets of reachable configurations.

An abstraction function is defined as a mapping $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall \mathcal{A} \in \mathbb{M}_\Sigma : \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\alpha(\mathcal{A}))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall \mathcal{A} \in \mathbb{M}_\Sigma : \mathcal{L}(\alpha'(\mathcal{A})) \subseteq \mathcal{L}(\alpha(\mathcal{A}))$. Given a term rewriting system τ and an abstraction α, a mapping $\tau_\alpha : \mathbb{M}_\Sigma \to \mathbb{M}_\Sigma$ is defined as $\forall \mathcal{A} \in \mathbb{M}_\Sigma : \tau_\alpha(\mathcal{A}) = \hat{\tau}(\alpha(\mathcal{A}))$ where $\hat{\tau}(\mathcal{A})$ is the minimal deterministic automaton describing the language $\tau(\mathcal{L}(\mathcal{A}))$. An abstraction α is *finitary*, if the set $\mathbb{A}_\Sigma$ is finite.

For a given abstraction function α, one can compute iteratively the sequence of automata $(\tau_\alpha^i(\mathcal{I}))_{i \geq 0}$. If the abstraction α is finitary, then there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(\mathcal{I}) = \tau_\alpha^k(\mathcal{I})$. The definition of the abstraction function α implies that $\mathcal{L}(\tau_\alpha^k(\mathcal{I})) \supseteq \tau^*(\mathcal{L}(\mathcal{I}))$.

If $\mathcal{L}(\tau_\alpha^k(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) = \emptyset$, then Problem (2.1) has a positive answer. If the intersection is non-empty, one must check whether a real or a spurious counterexample has been encountered. The spurious counterexample may be caused by the used abstraction (the counterexample is not reachable from the set of initial configurations). Assume that $\mathcal{L}(\tau_\alpha^k(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$, which means that there is a symbolic path:

$$\mathcal{I},\ \tau_\alpha(\mathcal{I}),\ \tau_\alpha^2(\mathcal{I}), \ldots, \tau_\alpha^{n-1}(\mathcal{I}),\ \tau_\alpha^n(\mathcal{I}) \tag{2.2}$$

such that $\mathcal{L}(\tau_\alpha^n(\mathcal{I})) \cap \mathcal{L}(\mathcal{B}) \neq \emptyset$.

Let $X_n = \mathcal{L}(\tau_\alpha^n(\mathcal{I})) \cap \mathcal{L}(\mathcal{B})$. Now, for each l, $0 \leq l < n$, $X_l = \mathcal{L}(\tau_\alpha^l(\mathcal{I})) \cap \tau^{-1}(X_{l+1})$ is computed. Two possibilities may occur: (a) $X_0 \neq \emptyset$, which means that Problem (2.1) has a negative answer, and $X_0 \subseteq \mathcal{L}(\mathcal{I})$ is a set of dangerous initial configurations. (b) $\exists m, 0 \leq m < n, X_{m+1} \neq \emptyset \wedge X_m = \emptyset$ meaning

that the abstraction function is too rough—one needs to refine it and start the verification process again.

In [BHRV06b], two general-purpose kinds of abstractions are proposed. Both are based on *automata state equivalences*. Tree automata states are split into several equivalence classes, and all states from one class are collapsed into one state. An abstraction becomes finitary if the number of equivalence classes is finite. The refinement is done by refining the equivalence classes. Both of the proposed abstractions allow for an automatic refinement to exclude the encountered spurious counterexample.

The first proposed abstraction is an *abstraction based on languages of trees of a finite height*. It defines two states equivalent if their languages up to the given height n are equivalent. There is just a finite number of languages of height n, therefore this abstraction is finitary. A refinement is done by an increase of the height n. The second proposed abstraction is an *abstraction based on predicate languages*. 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 $\mathcal{A} = (Q, \Sigma, F, q_0, \delta)$ be a tree automaton. Then, two states $q_1, q_2 \in Q$ are equivalent if the languages $\mathcal{L}(\mathcal{A}_{q_1})$ and $\mathcal{L}(\mathcal{A}_{q_2})$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$ provided that $\mathcal{A}_{q_1} = (Q, \Sigma, F, q_1, \delta)$ and $\mathcal{A}_{q_2} = (Q, \Sigma, F, q_2, \delta)$. Since there is just a finite number of subsets of $\mathcal{P}$, the abstraction is finitary. A refinement is done by adding new predicates, i.e. tree automata corresponding to the languages of all the states in the automaton of X_{m+1} from the analysis of spurious counterexample ($X_m = \emptyset$).

3 Forest Automata

In this chapter, we introduce *forest automata* which is a new formalism for representing sets of graphs. Our main motivation for creating this formalism has been verification of programs manipulating dynamically linked data structures. For this reason, in Section 3.1, we give an informal presentation of forest automata in the context of heaps (which may be viewed as a special kind of graphs) instead of the context of plain graphs. The way heaps are represented by forests will then be presented in more detail in Section 4.1. In Section 3.2, we formally define the notion of *hypergraphs* and their representation using forests. In Section 3.3, we discuss the representation of sets of forests using *forest automata*. Finally, Section 3.4 and Section 3.5 describe a hierarchical extensions of hypergraphs and forest automata respectively.

3.1 From Heaps to Forests

Now, we outline in an informal way our proposal of hierarchical forest automata and the way how sets of heaps can be represented by them (the more precise description of the encoding will be given in Section 4.1). 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 Figure 3.1 (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 contain

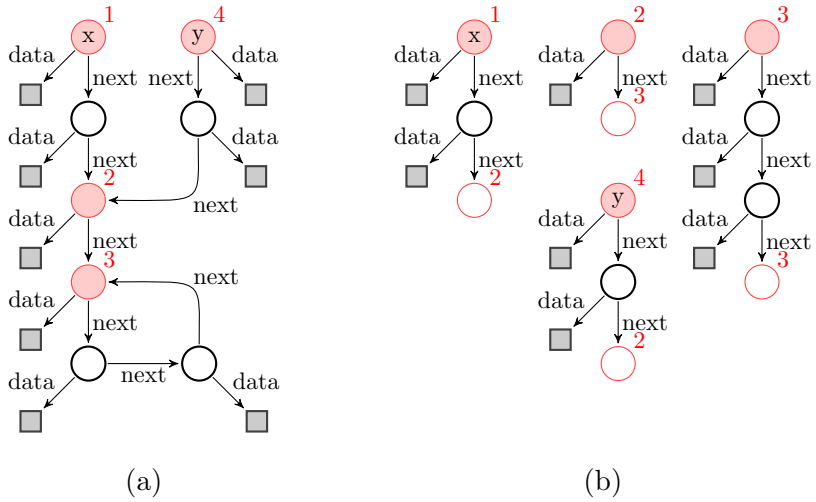


Figure 3.1: (a) A heap graph with cut-points highlighted in red, (b) the canonical tree decomposition of the heap with `x` ordered before `y`

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. Section 3.2). Thus, we represent unions of FA explicitly as *sets of FA* (SFA), which is similar to dealing with disjunctions of conjunctive

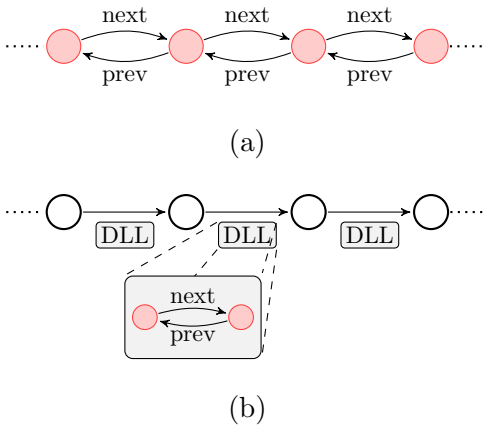


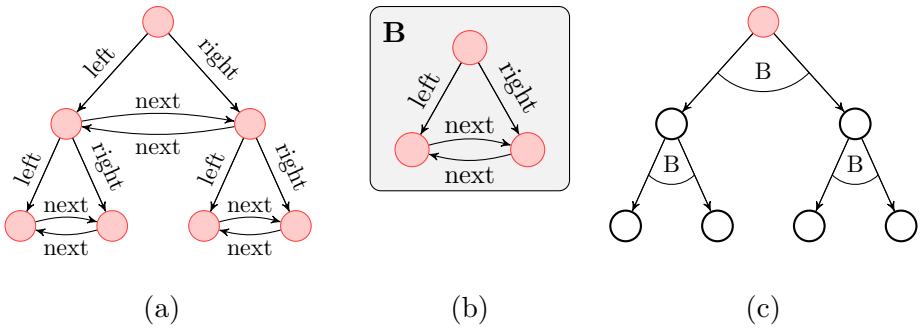
Figure 3.2: (a) A part of a DLL, (b) a hierarchical encoding of the DLL

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.[1] 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 3.2 (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 Figure 3.3. The tree with linked brothers from Figure 3.3 (a) is turned into a hypergraph with binary hyperedges shown in Figure 3.3 (c) using the box B from Figure 3.3 (b). The subgraph encoded by the box B can be connected to its surroundings via its input port i and *two* output ports $o1$, $o2$. Therefore, the hypergraph from Figure 3.3 (c) encodes it by a hyperedge with one source and *two* target nodes.

[1] 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.



(a) (b) (c)

Figure 3.3: (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.

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.[2] Intuitively, FA appearing in the alphabet of some superior FA play a role similar to that of inductive predicates in separation logic.[3] 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 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.

3.2 Hypergraphs and Their Representation

We now formalise the notion of hypergraphs and their forest representation.

[2]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.

[3]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.

3.2.1 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 $\overline{v}$ to denote a sequence $v_1, \ldots, v_n$ and $\overline{v}.i$ to denote its i^{th} 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 $\forall (v, a, \overline{v}), (v, a', \overline{v}') \in E$: $a = a' \implies \overline{v} = \overline{v}'$. 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 Section 3.1, 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 3.1 (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 Figure 3.3 (c).

3.2.2 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 $rr(T_i, j)$ the set of all root references to T_j in T_i, i.e., $rr(T_i, j) = \{v \in V_i \mid (v, j) \in E_i\}$. We also define $rr(T_i) = \bigcup_{j=1}^{n} rr(T_i, j)$.

A forest $F = (T_1, \ldots, T_n, R)$ *represents* the hypergraph $\otimes F$ 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 $\otimes F$, respectively. Finally, their ordering $\preceq_P$ is defined by the $\preceq_R$-ordering of the indices of the trees whose roots they are. Formally, $\otimes F = (V, E, P)$ where:

- $V = \bigcup_{i=1}^{n} V_i \setminus rr(T_i)$,
- $E = \bigcup_{i=1}^{n} \{(v, a, \overline{v}') \mid a \in \Gamma \wedge \exists (v, a, \overline{v}) \in E_i \ \forall 1 \leq j \leq \#(a) :$ if $\exists (\overline{v}.j, k) \in E_i$ with $k \in \{1, \ldots, n\}$, then $\overline{v}'.j = root(T_k)$, else $\overline{v}'.j = \overline{v}.j\}$,
- $I_P = \{root(T_i) \mid i \in I_R\}$,
- $O_P = \{root(T_i) \mid i \in O_R\}$,
- $\forall u, v \in I_P \cup O_P$ such that $u = root(T_i)$ and $v = root(T_j)$:

$$u \preceq_P v \iff i \preceq_R j.$$

3.2.3 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 $\otimes F$ *minimal* iff the roots of the trees $T_1, \ldots, T_n$ correspond to the *cut-points* of $\otimes F$, i.e., those nodes that are either ports, have more than one incoming hyperedge in $\otimes F$, 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 the minimal forest representation of G. 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 3.1 (b) shows a forest decomposition of the heap graph depicted in Figure 3.1 (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 Figure 3.1 (b) is a canonical representation of the heap graph of Figure 3.1 (a).

3.2.4 Root Interconnection Graphs

Let $F = (T_1, \ldots, T_n, R)$ be a forest. A *root interconnection graph* $\bigstar F = (V, E)$ is a (directed) graph in which the nodes $V = \{T_1, \ldots, T_n\}$ represent the roots of F, and the edges $E \subseteq V \times (\mathbb{N} \times \{1, 2\}) \times V$ represent the interconnection of the components of F. In particular, an edge labelled by $(k, 1)$ appears between T_i and T_j in $\bigstar F$ if and only if the DFT of T_i started in its root visits a reference to T_j after visiting $k - 1$ other root references (when not counting multiple occurrences of the same roots), and a reference to T_j is not visited anymore in the rest of the DFT. If a reference to T_j is visited after $k - 1$ other references (again not counting multiple occurrences of the same root), and it will be visited at least once more in the rest of the DFT (i.e., the root of T_j can be reached from the root of T_i via multiple paths, or, equivalently, $|rr(T_i, j)| > 1$), then, and only then $\bigstar F$ contains an edge connecting T_i and T_j labelled by $(k, 2)$.

Using the root interconnection graph, one can immediately see whether the corresponding forest is canonical or not. Indeed, a forest $F = (T_1, \ldots, T_n, R)$ is minimal if and only if each node in $\bigstar F$ is in $I_P \cup O_P$, has more than one incoming edge, or it has an incoming edge labelled by $(k, 2)$. Moreover, if a series of DFTs on $\bigstar F$ (which start from the nodes corresponding to input ports in I_P in the order given by $\preceq_P$, and in which an edge (k, i) is explored before (l, j) iff $k < l$) visits the nodes of $\bigstar F$ in the order $T_1, T_2, \ldots, T_n$, then F is canonical.

3.2.5 Manipulating the Forest Representation

In practice, it is often the case that one needs to modify the number of trees within a forest representation of a hypergraph. For instance, we can have an arbitrary forest representation and want to obtain a canonical one. Apart from changing the order of the trees, we might also need to eliminate certain roots (which are not cut-points) by gluing them (together with the trees rooted at them) with the leaves of other trees. For this reason, we define an additional

operation over forests which we call a *concatenation*. If a root of a tree T_j is referenced exactly once from a different tree T_i inside the forest (i.e., $|rr(T_i, j)| = 1$ and $|rr(T_k, j)| = 0$ for $k \neq i$) and it corresponds to neither an input nor an output port (i.e., $root(T_j) \notin I_R \cup O_R$), then T_j can be concatenated to T_i by replacing the leaf node referencing T_j in T_i by T_j. The concatenation of T_j to T_i within $F = (T_1, \ldots, T_n, R)$ is denoted by $concat(F, i, j)$.

Formally, let $v \in V_i$ such that $(v, j) \in E_i$ (i.e., v is a leaf of T_i representing the root reference to T_j). Assuming (w.l.o.g.) that $i < j$, $concat(F, i, j)$ is defined as

$$(T_1, \ldots, T_{i-1}, T'_i, T_{i+1}, \ldots, T_{j-1}, T_{j+1}, \ldots, T_n, R)$$

where $T'_i = (V'_i, E'_i)$ such that

- $V'_i = (V_i \setminus \{v\}) \cup V_j$
- $E'_i = (E_i \setminus \{(u, a, v) : u \in V_i \wedge a \in \Gamma\} \setminus \{(v, j)\}) \cup \{(u, a, Root(T_j)) : (u, a, v) \in E_i\} \cup E_j$

The operation *concat* preserves the semantics of the forest, therefore it holds that $\otimes concat(F, i, j) = \otimes F$ whenever such concatenation is possible.

3.3 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.

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 $\mathcal{L}_F(\mathcal{F}) = \{(T_1, \ldots, T_n, R) \mid \forall 1 \leq i \leq n : \ ot(T_i) \in \mathcal{L}(\mathcal{A}_i)\}$, 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 $\mathcal{L}(\mathcal{F}) = \{\otimes F \mid F \in \mathcal{L}_F(\mathcal{F})\}$.

3.3.1 Uniform and Canonicity Respecting FA

An FA $\mathcal{F}$ is called *uniform* if and only if for each forest $F \in \mathcal{L}_F(\mathcal{F})$, the hypergraph $\otimes F$ is well-connected, and for any two $F, F' \in \mathcal{L}_F(\mathcal{F})$, it holds that $\bigstar F = \bigstar F'$. Since all forests within a uniform FA are required to have the same root interconnection graph, the concatenation defined in Section 3.2.2 can also be easily performed on language of uniform FA (note that if some T_i does not correspond to a cut-point in F and it can be merged to T_j, then from the definition of uniform FA, any other F' is guaranteed to have some T'_i and T'_j such that T'_i can be merged into T'_j as well). Therefore, we can lift *concat* from single forests to an entire language of FA $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ as follows:

$$concat(\mathcal{L}(\mathcal{F}), i, j) = \{concat(F, i, j) : F \in \mathcal{L}(\mathcal{F})\}.$$

Moreover, an FA $\mathcal{F}'$ representing $concat(\mathcal{L}(\mathcal{F}), i, j)$ can easily be obtained from $\mathcal{F}$. In particular, assuming the sets of states of components $\mathcal{A}_i$ and $\mathcal{A}_j$ to be disjoint, we first replace each leaf transition in $\mathcal{A}_i$ labelled by the reference to j by all accepting transitions of $\mathcal{A}_j$ (we create new transitions by using the right-hand-side states of the leaf transitions of $\mathcal{A}_i$ and the symbol and the left-hand-side tuple of states of the accepting transitions of $\mathcal{A}_j$; in particullar, for a transition of $\langle j \rangle \to r$ of $\mathcal{A}_i$ and a transition $\alpha(q_1, \ldots, q_n) \to q$ of $\mathcal{A}_j$, we create a transition $\alpha(q_1, \ldots, q_n) \to r$). Then, we add all transitions of $\mathcal{A}_j$ into $\mathcal{A}_i$ and remove $\mathcal{A}_j$ from the resulting FA.

We say that an FA $\mathcal{F}$ *respects canonicity*[4] iff for each forest $F \in \mathcal{L}_F(\mathcal{F})$, the hypergraph $\otimes F$ 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, $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ iff $n = m$, $R = R'$, and $\forall 1 \leq i \leq n : \ \mathcal{L}(\mathcal{A}_i) \subseteq \mathcal{L}(\mathcal{A}'_i)$.*

Obviously, any canonicity respecting FA is also uniform. On the other hand, any uniform FA $\mathcal{F}$ can be easily transformed into a canonicity respecting one by first concatenating each redundant component[5] with its parent component

[4] We intentionally use the term *canonicity respecting FA* instead of *canonical FA* to stress that not all such FA are strictly canonical (this is due to the extensions that we will describe in Section 3.4 and Section 3.5). However, canonicity respecting FA can be treated as canonical for many practical purposes.

[5] As we have already mentioned in Section 3.2.3, a redundant component can be identified by looking for roots which are referenced only once.

(i.e., the component that contains the only reference to the given root) such that the resulting FA contains the minimal number of components. Then, we reorder the remaining components according to the DFT performed on the root interconnection graph.

3.3.2 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. As we have already mentioned in the Section 3.3, one can obtain a canonicity respecting FA from a uniform one. It remains to show how to convert an arbitrary FA into a set of uniform FA. In the following, we describe the computation which allow us to reconstruct the root interconnection graph of a given FA and which, if it is needed, allows us to split the FA into a set of uniform FA.

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 uniform. After that, some of the TA may get concatenated. 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 $t \in \mathcal{L}(q)$ 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 which 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. An example of the

$$
\begin{array}{ll}
\Delta\text{:} & \begin{array}{ccc} a(q,q) \to q, & a(r,q) \to q, & a(q,r) \to q, \\ a(s,s) \to q, & ref.1 \to r, & b \to s \end{array} \\
& \Downarrow \\
\Delta'\text{:} & \begin{array}{rr}
a(q_{(1,\{1\})}, q_{(1,\{1\})}) \to q_{(1,\{1\})}, & a(q_{(1,\emptyset)}, q_{(1,\emptyset)}) \to q_{(1,\{1\})}, \\
a(q_{(1,\{1\})}, q_{(1,\emptyset)}) \to q_{(1,\{1\})}, & a(q_{(1,\emptyset)}, q_{(1,\{1\})}) \to q_{(1,\{1\})}, \\
a(q_{(1,\{1\})}, r_{(1,\emptyset)}) \to q_{(1,\{1\})}, & a(r_{(1,\emptyset)}, q_{(1,\{1\})}) \to q_{(1,\{1\})}, \\
a(q_{(1,\{1\})}, q_{(\varepsilon,\emptyset)}) \to q_{(1,\{1\})}, & a(q_{(\varepsilon,\emptyset)}, q_{(1,\{1\})}) \to q_{(1,\{1\})}, \\
a(q_{(1,\emptyset)}, r_{(1,\emptyset)}) \to q_{(1,\{1\})}, & a(r_{(1,\emptyset)}, q_{(1,\emptyset)}) \to q_{(1,\{1\})}, \\
a(q_{(1,\emptyset)}, q_{(\varepsilon,\emptyset)}) \to q_{(1,\emptyset)}, & a(q_{(\varepsilon,\emptyset)}, q_{(1,\emptyset)}) \to q_{(1,\emptyset)}, \\
a(q_{(\varepsilon,\emptyset)}, r_{(1,\emptyset)}) \to q_{(1,\emptyset)}, & a(r_{(1,\emptyset)}, q_{(\varepsilon,\emptyset)}) \to q_{(1,\emptyset)}, \\
a(q_{(\varepsilon,\emptyset)}, q_{(\varepsilon,\emptyset)}) \to q_{(\varepsilon,\emptyset)}, & a(s_{(\varepsilon,\emptyset)}, s_{(\varepsilon,\emptyset)}) \to q_{(\varepsilon,\emptyset)}, \\
ref.1 \to r_{(1,\emptyset)}, & b \to s_{(\varepsilon,\emptyset)}
\end{array}
\end{array}
$$

Figure 3.4: Example of labelling (w, Y) obtained for a single component $A = (\Sigma, Q, \Delta, \{q_f\})$ during the transformation into a canonicity respecting FA (in the picture, labelled states are of the form $q_{(w,Y)}$). The newly obtained TA contains 3 different accepting states ($q_{(1,\{1\})}$, $q_{(1,\emptyset)}$, and $q_{(\varepsilon,\emptyset)}$) suggesting that the component needs to be split into 3. The language of $q_{(1,\{1\})}$ contains trees with two or more references to root 1. Similarly $q_{(1,\emptyset)}$ corresponds to trees having exactly one such a reference. Finally, trees within the language of $q_{(\varepsilon,\emptyset)}$ do not contain any reference at all.

labelling is depicted in Figure 3.4. 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[6]. 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 of the FA 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

[6]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.

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 w.r.t. 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 i^{th} tree was moved to the j^{th} 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.

3.3.3 Sets of FA

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 $\mathcal{L}_F(\mathcal{F}) = \{(a, b, R)\}$ and $\mathcal{L}_F(\mathcal{F}') = \{(c, d, R)\}$ 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 $\mathcal{L}(\mathcal{F}) \cup \mathcal{L}(\mathcal{F}')$.

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 $\mathcal{L}(\mathcal{S})$ 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.

3.3.4 Testing Inclusion on Sets of FA

The problem of checking inclusion on SFA, this is, checking whether $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{S}')$ 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 i^{th} child of the root is the root of a tree accepted by the i^{th} tree automaton of $\mathcal{F}$.

Trees accepted by $\mathcal{A}^{\mathcal{F}}$ are therefore unique encodings of hypergraphs in $\mathcal{L}(\mathcal{F})$. We will then test the inclusion $\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{S}')$ by testing the tree automata language inclusion between the union of TA obtained from $\mathcal{S}$ and the union of TA obtained from $\mathcal{S}'$.

Formally, let $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ be an FA where $\mathcal{A}_i = (\Sigma, Q_i, \Delta_i, F_i)$ for each $1 \leq i \leq n$. Without a loss of generality, assume that $Q_i \cap Q_j = \emptyset$ for each $1 \leq i < j \leq n$. We define the TA $\mathcal{A}^{\mathcal{F}} = (\Sigma \cup \{\curlywedge_n^R\}, Q, \Delta, \{q^{top}\})$ where:

- $\curlywedge_n^R \notin \Sigma$ is a fresh symbol with $\#(\curlywedge_n^R) = n$,
- $q^{top} \notin \bigcup_{i=1}^{n} Q_i$ is a fresh accepting state,
- $Q = \bigcup_{i=1}^{n} Q_i \cup \{q^{top}\}$, and
- $\Delta = \bigcup_{i=1}^{n} \Delta_i \cup \Delta^{top}$ where Δ^{top} contains the rule $\curlywedge_n^R(q_1, \ldots, q_n) \to q^{top}$ for each $(q_1, \ldots, q_n) \in F_1 \times \cdots \times F_n$.

It is now easy to see that the following proposition holds (in the proposition, "$\cup$" stands for the usual tree automata union).

Proposition 2 *For SCFA $\mathcal{S}$ and $\mathcal{S}'$,*

$$\mathcal{L}(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{S}') \iff \mathcal{L}(\bigcup_{\mathcal{F} \in \mathcal{S}} \mathcal{A}^{\mathcal{F}}) \subseteq \mathcal{L}(\bigcup_{\mathcal{F}' \in \mathcal{S}'} \mathcal{A}^{\mathcal{F}'}).$$

3.4 Hierarchical Hypergraphs

As discussed informally in Section 3.1, 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 (Figures 3.2 and 3.3). 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.

3.4.1 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[7]). 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.

3.4.2 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 $e = (v, a, \overline{v}) \in E$, 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

[7] 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.

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 $w = o_i, 1 \leq i \leq k$, $match(w) = \overline{v}.i$ (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 $E'' = (E \setminus \{e\}) \cup \{(v'', a', \overline{v}'') \mid \exists (v', a', \overline{v}') \in E' : \ match(v') = v'' \ \wedge \ \forall 1 \leq i \leq k : \ match(\overline{v}'.i) = \overline{v}''.i\}$.

We can now give an inductive definition of $[\![H]\!]$. Let $e_1 = (v_1, B_1, \overline{v}_1), \ldots, e_n = (v_n, B_n, \overline{v}_n)$ 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 3.2 (b) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)graph of Figure 3.2 (a). Similarly, Figure 3.3 (c) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)-graph of Figure 3.3 (a).

3.5 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 $\mathcal{L}(\mathcal{S})$ 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 $\mathcal{L}(\mathcal{S})$. 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 $G = (V, E, P) \in \mathcal{L}(\mathcal{F})$ 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 $(v, \mathcal{S}, \overline{v}) \in E$, labelled by a Γ-labelled hierarchical SFA $\mathcal{S}$, by the edge $(v, \mathcal{L}(\mathcal{S})^\sharp, \overline{v})$, labelled by the box $\mathcal{L}(\mathcal{S})^\sharp$ where $\mathcal{L}(\mathcal{S})^\sharp$ denotes the set (box) $\{X^\sharp \mid X \in \mathcal{L}(\mathcal{S})\}$. Then, we define

the semantics of a hierarchical FA $\mathcal{F}$ over Γ as the set of Γ-labelled (plain) hypergraphs $[\![\mathcal{F}]\!] = [\![\mathcal{L}(\mathcal{F})^\sharp]\!]$.

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 $\mathcal{L}(\mathcal{S})^\sharp$ 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 $\mathcal{L}(\mathcal{S})^\sharp$ 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 $\mathcal{L}(\mathcal{S})^\sharp$.

Hierarchical SFA are therefore finite representations of sets of hypergraphs with possibly unbounded numbers of cut-points.

3.5.1 On Well-Connectedness of Hierarchical FA

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 well-connectedness and inclusion 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. Moreover, 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.

3.5.2 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 $i \in br(C)$.[8]

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 in the following paragraph.

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 $[\![\mathcal{S}]\!]$, 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.

3.5.3 Checking Properness and Well-Connectedness

We now outline algorithms for checking properness of nested SFA and well-connectedness of SFA.

[8] Notice that this definition is correct since boxes of level 0 have no nested boxes, and the recursion stops at them.

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 Section 3.3.2. 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 $\mathcal{L}(\mathcal{F})$ without a need of reasoning about the inner structure of the nested SFA.

Let us have a canonicity respecting SFA S of some level such that its nested SFA are proper and we know the value of the function br for all of them.[9] We assume that S contains at least one FA, and that the languages of all $\mathcal{F} \in S$ are nonempty.

Moreover, to make the algorithms of checking properness and box-connectedness faster and simpler, we exploit the fact that the algorithm we describe in Section 3.5.7 in fact produces automata respecting canonicity in a somewhat stronger sense than described in Section 3.5.1. We define the stronger notion of respecting canonicity below.

Given $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R) \in S$, a *path* in a tree $t \in \mathcal{L}(\mathcal{A}_i)$ from v to w, $v, w \in dom(t)$, is a sequence $v = v_0, (a_1, k_1), v_1 \ldots, (a_m, k_m), v_m = w, 0 \leq m$, where for each $1 \leq i \leq m$, v_i is the k_i-th son of v_{i-1} and $t(v_i) = a_i$. The path is *backward passable* iff for each $1 \leq i \leq m$, the label a_i is backward passable at the k_i-th position, which means that there is a proper nested SFA $S_i \in a_i$ and $j \in br(S_i)$ such that $j + \sum_{\{b \in a_i | b \preceq_\Gamma S_i, b \neq S_i\}} \#(b) = k_i$.

For each $1 \leq i \leq n$ and each $t \in \mathcal{L}(\mathcal{A}_i)$, we define the *reachability relation*[10] $\rho_i^t \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ on the roots of $\mathcal{F}$ that contains a pair (j, k) iff one of the following three conditions holds:

[9] Notice that even though respecting canonicity assumes properness of nested SFA and we require here proper SFA to be canonicity respecting, this is not a circular dependency. For an SFA to respect canonicity, we only require its nested SFA to be proper. So, for an SFA of level j to respect canonicity, we require properness of SFA of level $j-1$ only. Respecting canonicity in an SFA of level 0 does not depend on the notion of properness since SFA of level 0 have no nested SFA. Properness on level j then depends on respecting canonicity on level j.

[10] Notice, that the reachability relation is related to the root interconnection graph which is defined in Section 3.2.4. Unlike reachability relation, the root interconnection graph does not record backward passable paths. On the other hand, it also contains some information about the number of paths between nodes.

1. $i = j$ and there is a leaf v of t with $t(v) = k$, or

2. $i = k$, there is a leaf v of t with $t(v) = j$, and the path from the root of t to v is backward passable, or

3. there are nodes u, v, w of t such that both v and w are leaves of the subtree rooted by u, $t(v) = j$, $t(w) = k$, and the path from u to v is backward passable.

We say that $\mathcal{F}$ is an *FA with uniform reachability* iff for each $1 \leq i \leq n$, ρ_i^t is the same for all $t \in \mathcal{L}(\mathcal{A}_i)$. If it is the case, then we denote the reachability relation as ρ_i. We further say that an SFA S *strongly respects canonicity* iff it respects canonicity as defined in Section 3.5.1 and all its elements are FA with uniform reachability. In Section 3.5.7, we show the transformation of FA into SFA that strongly respect canonicity and we also show how to compute the relation ρ_i.

If $\mathcal{F}$ is an FA with uniform reachability, we define the *global reachability relation* $\rho = (\bigcup_{1 \leq i \leq n} \rho_i)^*$ on roots of $\mathcal{F}$.[11] Notice that $(i, j) \in \rho$ iff for all $H \in [\![\mathcal{L}_H(\mathcal{F})]\!]$ there are two nodes u and v that correspond to the i-th and j-th root of $\mathcal{F}$, respectively, and such that there is a path from u to v in H.

Properness of an SFA representing some box and computing br on it is then done as follows. First, a singleton SFA $\{\mathcal{F}\}$ with ι being the only input port of $\mathcal{F}$ is proper iff (1) for all $1 \leq i \leq n$, $(\iota, i) \in \rho$ and (2) for all $o, o' \in O_R$, $(o, o') \in \rho \implies (o', \iota) \in \rho$. If $\{\mathcal{F}\}$ is proper, then $br(\{\mathcal{F}\})$ equals the set $\{o \in O_R \mid (o, \iota) \in \rho\}$. Finally, assuming that an SFA S strongly respects canonicity, S is proper iff all its elements agree on the values of I_R and O_R, and all the singleton SFA $\{\mathcal{F}\}$, $\mathcal{F} \in S$, are proper and agree on the value of $br(\mathcal{F})$. This value then equals $br(S)$.

Box-connectedness of an SFA S that strongly respects canonicity and that has proper nested SFA for which we know the values of br can be checked similarly as properness, i.e., using the relation ρ. Particularly, S is box-connected if and only if for all $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R) \in S$ and for all $1 \leq k \leq n$ there is some $\iota \in I_R$ such that $(\iota, k) \in \rho$.

3.5.4 On Inclusion of 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 order to be able to use our in practice (see Chapter 4), 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 $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}') \implies \mathcal{L}(\mathcal{F})^\sharp \subseteq \mathcal{L}(\mathcal{F}')^\sharp \implies [\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ obviously hold, we may safely approximate the solution of the inclusion problem by deciding whether $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ (i.e., we abstract away the semantics of nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ and treat them as ordinary labels).

[11] Here, the * stands for the reflexive and transitive closure.

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 Section 3.3, based on a canonical forest representation and canonicity respecting FA, since they rely on well-connectedness of hypergraphs from $\mathcal{L}(\mathcal{F})$ and $\mathcal{L}(\mathcal{F}')$, 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 $\mathcal{L}(\mathcal{F})$ or $\mathcal{L}(\mathcal{F}')$ 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.

3.5.5 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 $G = \otimes F$ which is box-connected and such that both G and G^{br} are deterministic. By extending the approach used in Section 3.3, 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 such that they 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 w.r.t. 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 Section 3.5.7.

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 $F \in \mathcal{L}_F(\mathcal{F})$ is a canonical representation of the hypergraph $\otimes F$. We abbreviate canonicity respecting hierarchical FA as hierarchical CFA. Analogi-

cally 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,* $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ *iff* $n = m$, $R = R'$, *and* $\forall 1 \leq i \leq n : \mathcal{L}(\mathcal{A}_i) \subseteq \mathcal{L}(\mathcal{A}'_i)$.

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. Section 4.6 in Chapter 4). 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, as described in the Section 3.5.7.

3.5.6 Precise Inclusion on Hierarchical FA

In many practical cases, approximating the inclusion $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ by deciding $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ is actually precise. A condition that guarantees this is the following:

Condition 1. $\forall H \in \mathcal{L}(\mathcal{F})^\sharp \, \forall H' \in \mathcal{L}(\mathcal{F}')^\sharp : H \neq H' \implies [\![H]\!] \cap [\![H']\!] = \emptyset$. 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 $H \in \mathcal{L}(\mathcal{F})^\sharp \cup \mathcal{L}(\mathcal{F}')^\sharp$ 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 the Chapter 4 satisfy Conditions 2 and 3, and so Condition 1 is satisfied too. Hence, inclusion tests performed within our case studies are precise.

3.5.7 Transforming Hierarchical FA into Canonicity Respecting Hierarchical FA

The labelling considered in Section 3.3.2 when transforming (non-hierarchical) FA into sets of canonicity respecting FA does not cover the cases when the nodes which are the roots of some tree components are reachable from nodes pointed by program variables only when considering backward reachability through boxes. We solve this problem by extending the labelling from Section 3.3.2 as described below. Consider a hierarchical forest automaton $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$, $n \geq 1$, with a set X of nested SFA that are proper and its component tree automaton $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$, $1 \leq i \leq n$. We label each $q \in Q_i$ by a set of extended labels (w, Y, Z_1, Z_2), $w \in \{1, \ldots, n\}^*$, $Y \subseteq \{1, \ldots, n\}$, $Z_1 \subseteq \{1, \ldots, n\} \times X \times \mathbb{N}$, and $Z_2 \subseteq \{1, \ldots, n\} \times X \times \mathbb{N} \times \{1, \ldots, n\}^*$ for which there is a tree $t \in \mathcal{L}(q)$ such that

- w and Y are as in the non-hierarchical case.
- $(r, S, i) \in Z_1$ iff there is a backward passable[12] path from a leaf labelled by the root reference r to the root of the tree t, and this leaf is an S_i-successor of some node in the unordered tree t' where $ot(t') = t$.
- $(r, S, i, w') \in Z_2$ records the fact that in the tree t, there is a subtree t' with the root labelled by (w', Y', Z_1', Z_2') such that $(r, S, i) \in Z_1'$. This labelling is used to resolve cases where there is a backward passable path from a root reference into some intermediate node in the tree, but not to the root of the tree t.

Such labelling can be obtained in a similar way as in the case of non-hierarchical automata—i.e., by first labelling states w.r.t. the leaf rules and then propagating the so far obtained labels bottom-up. Elements of the Z_1 sets are not propagated when a transition rule reads an edge without backward reachability at the concerned position. 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 accepting trees with the appropriate label of the root (in a similar way as in Section 3.3.2).

The extended labels guarantee that each FA $\mathcal{F}$ obtained above is an FA with *uniform reachability* (see Section 3.5.2). The relation ρ_i can be derived directly from the label of the final states of $\mathcal{A}_i$. Particularly, if the label is (w, Y, Z_1, Z_2), then $(j, k) \in \rho_i$ iff:

1. $i = j$ and k appears in w, or
2. $i = k$ and $(j, S, l) \in Z_1$ for some S and l, or
3. $(j, S, l, w') \in Z_2$ for some S and l such that k appears in w'.

Clearly, each of the FA created above represents a set of hierarchical hypergraphs that have the same number of roots. However, as in the case of non-hierarchical hypergraphs, some roots need not correspond to cut-points. This problem is solved in the same way as in non-hierarchical case.

[12] See Section 3.5.3 for the definition of a backward passable path.

Forward Traversal. In order to transform each FA $\mathcal{F}$ obtained above into the (strongly) canonicity respecting form, its component TA are subsequently ordered according to the depth-first traversal on the root interconnection graph extended for hierarchical FA (see Section 3.2.4) such that nodes again correspond to the roots of the forest representation of the hypergraphs encoded by $\mathcal{F}$, and edges represent the reachability relation $\bigcup_{1 \leq i \leq n} \rho_i$. The edges of the extended root interconnection graph are labelled by natural numbers using the extended labels as described below. Successors of nodes in the root interconnection graph graph are then explored according to these numbers in the depth-first traversal on the graph.

Let us denote by $x_{\mathcal{A}}$ the node of the root interconnection graph corresponding to a component TA $\mathcal{A}$ (i.e., to the roots of the trees accepted by $\mathcal{A}$). For each component TA $\mathcal{A}$, assuming that its final states are labelled by (w, Y, Z_1, Z_2), we label the edges leading from $x_{\mathcal{A}}$ to the nodes corresponding to TA referenced from w by natural numbers assigned in the order given by w. Then, for each component TA $\mathcal{A}$ of $\mathcal{F}$ whose labelling of final states contains a Z_1 triple (r, S, i) where r references a TA $\mathcal{A}'$, we label the edge leading from $x_{\mathcal{A}'}$ to $x_{\mathcal{A}}$ by a number assigned to the pair (S, i) in the lexicographic ordering on all such pairs that appear in the Z_1 triples of the labels of the component TA of $\mathcal{F}$. (We use numbers greater than those used in the previous phase of numbering). Finally, for each label (r, S, i, w') of the final states of some component TA $\mathcal{A}$, the TA $\mathcal{A}'$ referenced by r, and each TA $\mathcal{A}''$ referenced from w', we label the edge leading from $x_{\mathcal{A}'}$ to $x_{\mathcal{A}''}$ by a number obtained from the lexicographic ordering of the triples (S, i, r') where r' ranges over references that appear in the w' parts of the Z_2 labels. (We again use numbers greater than those used in the previous phases of numbering.) If multiple numbers are assigned to a single edge, the smallest is chosen.

Backward Traversal. The just described ordering of the component TA of a given FA based on the root interconnection graph orders the component TA in a way consistent with the order $\preceq_H$ that is induced on the root-points of the represented hypergraphs by the further described deterministic depth-first traversal on the corresponding backward reachability hypergraphs H^{br}. In particular, the corresponding DFT on the backward reachability hypergraphs starts from the input ports and it is driven by the fixed ordering on the input ports and the labels of hyperedges. The ordering of the inverted hyperedges (labelled by symbols from X^{br}) is inherited from the ordering of the hyperedges on which they are based and from the number of the output port used. Moreover, the DFT normally explores first original hyperedges and only then the inverted hyperedges. However, the inverted hyperedges are prioritised whenever the traversal comes to a node x using an inverted hyperedge and x is not a root of some tree. In such a case, the DFT continues the search first by inverted hyperedges and only then by the regular hyperedges.

The above described handling of the inverted hyperedges forces the DFT on backward reachability hypergraphs to reach a root of a tree component before alternating the direction of the DFT inside the tree component. We

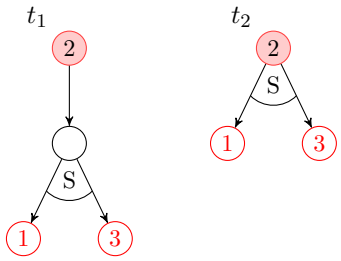


Figure 3.5: Two tree components problematic for handling the order of roots

explain the reason behind this on an example. Suppose that we have a TA accepting tree components t_1 and t_2 rooted at the root-point 2 as depicted in Figure 3.5. Suppose that all the edges in these trees are backward passable and that the accepting state of this automaton is labelled by the following label $(\text{“13”}, \{1,3\}, \{(1,S,1),\ (3,S,2)\},\ \{(1,S,1,\text{“13”}), (3,S,2,\text{“13”})\})$. Further, assume that the root-point r_1[13] is the only input port of the represented hypergraphs. Hence, the DFT on the described trees will start from the reference to r_1 (represented by the node labelled by 1 in Figure 3.5). In the backward reachability hypergraph based on t_1, a DFT without the described priority of inverted hyperedges would go from r_1 to the internal node of t_1 using the inverted hyperedge $(S,1)$ and then it would continue back to r_1, backtrack, and then go to r_3 via the different output ports of S. So, the induced ordering of the root-points would be $1 \preceq_H 3 \preceq_H 2$. On the other hand, in the backward reachability hypergraph based on t_2, such a DFT would go from r_1 to r_2 and then to r_3, giving a different induced ordering of the root-points, namely $1 \preceq_H 2 \preceq_H 3$. The described DFT forces the search in the backward reachability hypergraph based on t_1 to continue from the internal node of t_1 by an inverted hyperedge to r_2 and only after that to continue to r_3. So, the induced ordering is the same in both the cases. Note that the same ordering is obtained by the DFT on the root interconnection graph where the edge from r_1 to r_2 has a bigger priority than the edge from r_1 to r_3.

Forward-Unreachable Components. The FA obtained after splitting based on the extended labels described above, removing the redundant roots, and re-arranging the particular TA according to the root interconnection graph are still not necessarily canonicity respecting. Respecting of canonicity is not guaranteed in cases when there exist *root-points of Type 3* in the represented hypergraphs—i.e., when there exist loops without any incoming edge in these hypergraphs. This situation can easily be detected by looking for a component TA accepting a tree representation of such loops. In particular, this amounts to looking for a component TA accepting trees such that (1) their roots are not ports in the forest representation, (2) they are not referred from the trees accepted by any other TA—this can easily be checked by inspecting the label w of the final states of the other TA, and (3) they contain a single leaf with

[13] Let us denote the i-th root point as r_i.

a root reference to itself—this can be checked by inspecting the labels w and Y of the accepting states of the concerned TA. The canonisation procedure then rearranges the concerned TA such that the root of each accepted tree is the smallest node according to the above described depth-first traversal on H^{br} (i.e., the node that should be the canonically chosen root-point of Type 3). Intuitively, this amounts to a rotation of the concerned backward reachable loops represented by tree automata rules so that the nodes that are identified as the root-points become the roots of the tree representation.

More formally, let $\mathcal{A}_r$ be the TA that we need to rotate, r being the number of $\mathcal{A}_r$ in the concerned FA, and let $(w^{\mathcal{A}_r}, Y^{\mathcal{A}_r}, Z_1^{\mathcal{A}_r}, Z_2^{\mathcal{A}_r})$ be the label associated with the accepting states of $\mathcal{A}_r$. The states of $\mathcal{A}_r$ that accept the nodes that should become the new roots are identified as follows. Let k be the number of the root-point from where the DFT on the represented hypergraphs comes (without passing through any other root-points) to the nodes corresponding to the roots of the tree components represented by $\mathcal{A}_r$. Identifying k is easy since $x_{\mathcal{A}_k}$ is the predecessor of $x_{\mathcal{A}_r}$ in the DFT performed on the root interconnection graph. The edge from $x_{\mathcal{A}_k}$ to $x_{\mathcal{A}_r}$ exists in the root reference reachability graph due to existence of a backward passable path from the root of each tree represented by $\mathcal{A}_r$ to a root reference to k. Existence of this path is captured by the label $Z_1^{\mathcal{A}_r}$, concretely by an element $(x, S, i) \in Z_1^{\mathcal{A}_r}$.[14]

Now, the TA $\mathcal{A}_r = (Q_r, \Sigma, \Delta_r, F_r)$ can be rotated as described in the following. Let $R = (\ldots, q_{i_1}, \ldots, q_{i_2}, \ldots) \xrightarrow{a} q \in \Delta$ be a rule such that the w label of q_{i_1} contains r, and the Z_1 label of q_{i_2} contains (k, S, i).[15] Such a rule appears exactly once in each run of $\mathcal{A}$ since a reference to r may appear only once at the leaf level (otherwise we would not be dealing with a root-point of Type 3), and also S_i can link with a single leaf only (otherwise we would obtain a nondeterministic hypergraph). If there are more rules like R in $\mathcal{A}_r$, then we may split $\mathcal{A}_r$ to several automata containing a single rule of the described kind, and process each of the automata separately (which we assume to be the case in the following). When we transform $\mathcal{A}_r$ to $\mathcal{A}'_r$ that accepts trees in which the concerned loops are represented in the appropriately rotated way, the rule R is redirected to a newly introduced accepting state, the rules that used to originally lead to accepting states are redirected to states originally reading a reference to r, and reading a reference to r while going to q is allowed. Formally, for some $q_{fin} \notin Q_r$, $\mathcal{A}'_r = (Q_r \cup \{q_{fin}\}, \Sigma, \Delta'_r, \{q_{fin}\})$ where $\Delta' = (\Delta_r \setminus (\{R\} \cup \{\xrightarrow{r} q\})) \cup \Delta''$. The set of the newly added rules is defined as $\Delta'' = \{(\ldots, q_{i_1}, \ldots, q_{i_2}, \ldots) \xrightarrow{a} q_{fin}\} \cup \{(q_1, \ldots, q_k) \xrightarrow{b} q_r \mid (q_1, \ldots, q_k) \xrightarrow{b} q_f \in \Delta_r, q_f \in F_r, \xrightarrow{r} q_r \in \Delta_r\} \cup \{\xrightarrow{r} q\}$.

As a consequence of the rotation, the final state of the rotated TA may have a different label (w, Y, Z_1, Z_2) than the original one.[16] This may cause that the FA with the new TA inside has a different root interconnection graph and

[14] If there are more backward passable paths from the roots of the trees represented by $\mathcal{A}_r$ to a root reference to k, then each such path has a different record in $Z_1^{\mathcal{A}_r}$. In such a case, we choose the one with the smallest (S, i).

[15] Note that q_{i_1} and q_{i_2} can appear swapped on the left-hand side of the rule too.

[16] The order of root references in the string w can be different, and in each $(k, S, i, w') \in Z_2$, w' can be ordered differently as well. Y and Z_1 stay unchanged.

hence different ordering of the roots. Therefore after each TA rotation, we recompute the root reachability graph and reorder the forest. Note that the order of the roots that are originally ordered before the root of the rotated TA is not affected. Therefore, even if there are more root-points of Type 3, the rotations on each of them will be done at most once, and hence the canonisation procedure terminates.

As we have already mentioned before, the described procedure yields FA that strongly respect canonicity which allows us to check properness and box-connectedness by computing the reachability relation on the roots of the FA.

3.6 Conclusions and Future Work

With the aim of obtaining a formalism suitable for representing sets of heaps, we have introduced forest automata as a formalism for representing sets of graphs, decomposed into tree components. We have discussed various properties of this formalism. First of all, we have shown how one can obtain a canonical representation of a given set of graphs. Furthermore, we have shown that the language inclusion of canonical forest automata can be efficiently decided by translating them into tree automata. In addition, we have proposed a hierarchical extension of forest automata, which on one hand greatly increases the expressive power of our formalism, but on the other hand, certain operations—such as language inclusion checking—become more complicated. For this reason, we have also demonstrated that these operations can be safely approximated. Experiments with the verification approach based on FA that we present in the next chapter show that the proposed notion of FA can be quite useful in practice. These results also indicate that the approximate inclusion checking is sufficient in practice.

An interesting area for the future work which has not been investigated so far is a characterisation of the class of graphs (heaps) which can be described by hierarchical forest automata. Moreover, even though our experiments show that the approximate inclusion checking on hierarchical FA that we have proposed is quite successful in practice, it would be interesting to know whether (precise) inclusion checking on FA is decidable (and efficiently implementable). A somewhat related problem, which we will come across in the next chapter, is then the problem of computing intersections of FA (which we will also approximate in Section 4.5). Finally, one can also consider extending FA by recursively nested boxes. They would greatly increase the expressive power of the formalism, however, it is so far unclear how to implement the required algorithms over such an extension.

4 Forest Automata-based Verification

In this chapter, we build on the notion of forest automata, and we propose a forest-automata-based verification procedure for sequential programs manipulating complex dynamically linked data structures. We concentrate on programs manipulating various forms of singly- and doubly-linked lists (SLL/DLL), possibly cyclic, shared, hierarchical, and/or having various additional selectors (e.g., head pointers, tail pointers, data, etc.), as well as various forms of trees. We, in particular, consider C pointer manipulation, but our approach can be easily applied to any other similar language. We focus 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.

As we have sketched already at the beginning of Chapter 3, in our forest-automata-based representation, a heap is split in a canonical way into several tree components such that roots of the trees correspond to cut-points. The tree components can refer to the roots of each other. Using this decomposition, sets of heaps with a bounded number of cut-points are then represented by forest automata. Moreover, we allow alphabets of FA to contain nested FA, allowing us to also represent sets of heaps with an unbounded number of cut-points, which is necessary in many practical cases (e.g., when one deals with sets of DLLs). Finally, since FA are not closed under union, we work with sets of forest automata.

A fundamental property of our newly proposed formalism of forest automata is that C program statements manipulating pointers can be easily encoded as operations modifying FA. Due to this and due to the fact that FA are based on tree automata, we can build on the concept of abstract regular tree model checking (see [BHRV06b] or Section 2.5) to obtain a new symbolic verification procedure for the considered class of programs. The procedure then works as follows: The algorithm maintains a set of visited program configurations and a set of program configurations which need to be processed. At the beginning, the set of visited program configurations is empty, and the set of program configurations waiting to be processed contains the initial configuration of the program to be analysed which consists of the initial assignment of program variables, the empty heap, and the program counter pointing to the first instruction of the program. Then, the algorithm iteratively picks one waiting program configuration and performs a symbolic execution of the appropriate program statement. This essentially means that one takes a forest automaton representing a set of heaps and transforms it into a new forest automaton. The set of heaps represented by the newly obtained forest automaton reflects the

change within the heap caused by the execution of the given program statement. In addition to that, one can also apply *abstraction* in order to be able to obtain sets of all reachable configurations, which are typically infinite, in a finite number of steps. In the next step, the algorithm checks whether the newly created program configuration is covered by the set of already visited program configurations by means of testing inclusion of languages represented by forest automata. If the newly obtained symbolic configuration is not covered by the set of visited program configurations, it is inserted into the set of waiting program configurations. The process then continues by picking another waiting configuration. During the symbolic execution, the algorithm checks whether the verified code behaves properly, i.e., it does not dereference invalid pointers, it does not produce memory leaks, etc. If the program does not operate properly, the procedure is immediately terminated, and an error is reported. If the set of waiting configurations becomes empty, the procedure terminates and outputs that the program is safe.

When an error is encountered, it remains to find out whether it is reachable within the original program, or it was encountered due to an excessive abstraction. In order to check, whether the error is indeed reachable, one can execute the corresponding trace without the abstraction. If such trace cannot be executed, then the set of reachable program configurations is over-approximated too much, and the abstraction needs to be refined. The refinement can be done globally which is, however, not very efficient. A better solution is to use the counterexample-guided abstraction refinement as introduced in the framework of abstract regular tree model checking (see again [BHRV06b] or Section 2.5). For that to work, one needs to be able to execute the error trace backwards which is discussed later on.

Our approach has been implemented in a prototype tool called *Forester* as a `gcc` plug-in. This allows us to demonstrate that the proposed approach is very promising as the tool can successfully handle multiple highly non-trivial case studies (for some of which we are not aware of any other tool that could handle them fully automatically).

Plan of the Chapter. The rest of this chapter is organised as follows. In Section 4.1, we describe how we perform symbolic execution of C program statements over FA. Section 4.2 provides the main loop of our verification procedure. In Section 4.3, we present an algorithm for automatic discovery of nested FA. The problem of abstraction is described in Section 4.4. Section 4.5 discusses how an error trace suspected to lead to an error can be executed backwards in order to obtain an abstraction refinement if the error is spurious. In Section 4.6, we experimentally evaluate the performance of our prototype tool based on FA. Section 4.7 contains information about related work. Finally, Section 4.8 concludes the chapter.

4.1 Symbolic Execution

In this section, we make more precise the way we encode heaps using FA (refining the main idea sketched in Section 3.1 when motivating the notion of FA), and we describe the symbolic execution of C programs over sets of hierarchical FA, which is at the heart of our verification procedure. We consider sequential, non-recursive C programs manipulating dynamically linked data structures via the following program statements:

- `x` = `y`,
- `x` = `y->s`,
- `x` = `null`,
- `x->s` = `y`,
- `x->s` = `d` where `d` is a constant of data domain,
- `malloc(x)`, and
- `free(x)`

together with pointer and data equality tests and common control flow statements as discussed in more details below[1]. Each allocated cell may have several next pointer selectors and also selectors containing data from some finite domain[2]. 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.

4.1.1 Executing C Statements on Hypergraphs

A single heap configuration can be viewed as a deterministic $(Sel \cup Data \cup Var)$-labelled hypergraph with the ranking function being such that $\#_e(x) = 1 \Leftrightarrow x \in Sel$ and $\#_e(x) = 0 \Leftrightarrow x \in Data \cup Var$. In the hypergraph, the nodes represent allocated memory cells. Selectors are represented by the unary hyperedges labelled by elements of *Sel*. The nullary hyperedges labelled by elements of $Data \cup Var$ represent data values and program variables[3]. 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`.

The symbolic computation of reachable heap configurations is done over the control flow graph (CFG) obtained from the program under verification. A control flow action a, corresponding to one of the supported statements, applied to a hypergraph G (i.e., to a single configuration) returns a hypergraph $a(G)$ that is obtained from G as follows:

[1] Most C statements for pointer manipulation can be translated the listed statements, including most type casts. The problem of restricted pointer arithmetic is discussed in Section 4.1.4.

[2] No abstraction for such data is considered.

[3] 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.

- 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 variables aliased to `x` with `undef`), and `x->s` $= \mathtt{d}_{new}$ 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.

4.1.2 Executing C Statements on FA

We, however, compute not on single hypergraphs representing particular heaps but on sets of them represented by sets of canonicity respecting hierarchical FA (SCFA). Below, we first refine the description of the way we represent sets of heaps via FA (sketched in Section 3.1). Then, we present the different steps in which we symbolically execute C statements over FA. Next, we explain how pointer updates can be applied to it. Finally, we provide an example of the symbolic execution.

Heap Encoding Details. As described in Chapter 3, in order to represent sets of heaps using SCFA, we first decompose a $(Sel \cup Data \cup Var)$-labelled hypergraph into a forest of unordered, $(Sel \cup Data \cup Var)$-labelled trees. Then, the forest of unordered, $(Sel \cup Data \cup Var)$-labelled trees is transformed into a forest of ordered, $(Sel \cup Data \cup Var)^*$-labelled trees in which the alphabet $(Sel \cup Data \cup Var)^*$ is ranked using the ranking function $\#_v$ which is defined recursively as follows:

$$\begin{array}{lcl} \#_v(\varepsilon) & = & 0 \\ \#_v(x\alpha) & = & \#_e(x) + \#_v(\alpha) \end{array}$$

In addition to that, we also decouple the selectors and their values (i.e., the values are stored within dedicated nodes) in order to handle ordinary and pointer data in a uniform way (selectors are always represented by a graph edge, no matter what type of data they contain). As a result, we obtain a forest of trees in which a node can be labelled by:

- $(Sel \cup Var)^*$ if it is a root,
- Sel^* if it is an internal node, or
- $Data$ if it is a leaf (we consider that the set $Data$ also contains all possible root references).

Phases of Symbolic Execution. In order to be able to implement the symbolic execution on SCFA along the edges of a given CFG, we need to be able to compute the following. 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.

We derive the SCFA $x(\mathcal{S})$ from $\mathcal{S}$ in several steps in which we process each CFA $\mathcal{F} \in \mathcal{S}$ separately. 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 roots. In the next phase, we introduce *auxiliary roots* for every node referred to by x (as discussed in Section 4.1.3) to the forest representation. Third, we perform the *actual update*, which due to the previous step amounts to manipulation with roots only. 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 roots.

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). Conversely to unfolding, the folding is done by replacing the part of the top-level SFA by a single rule labelled by the nested SFA. More details on how we actually perform the folding are provided in Section 4.3.

Pointer Updates. Before performing the actual update, we check whether the operation tries to access the successors of `null` or `undef` nodes in which case we have encountered a null or undefined pointer dereference. Otherwise, we continue by performing one of the following actions, depending on the particular statement:

- `x = y`, `x = null`: `x` is removed from all labels in which it appears. Then we label by `x` each transition that is labelled by `y` (or by `null`, respectively).
- `malloc(x)`: `x` is removed from all labels in which it appears. Then a new tree automaton accepting exactly a single tree encoding a single heap node pointed by x and having undefined successors is added.
- `x = y->s`: `x` is removed from all labels in which it appears. Then the TA which accepts nodes pointed by `y` is split (as described in Section 4.1.3 below) at the selector `s`, and the transitions of the newly created TA leading to its accepting state, at which `y->s` is accepted, are labelled by `x`.
- `x->s = y`: The TA that accepts nodes pointed by variable `x` is split at the selector `s`. Provided that q_f is an accepting state and q_s accepts a reference

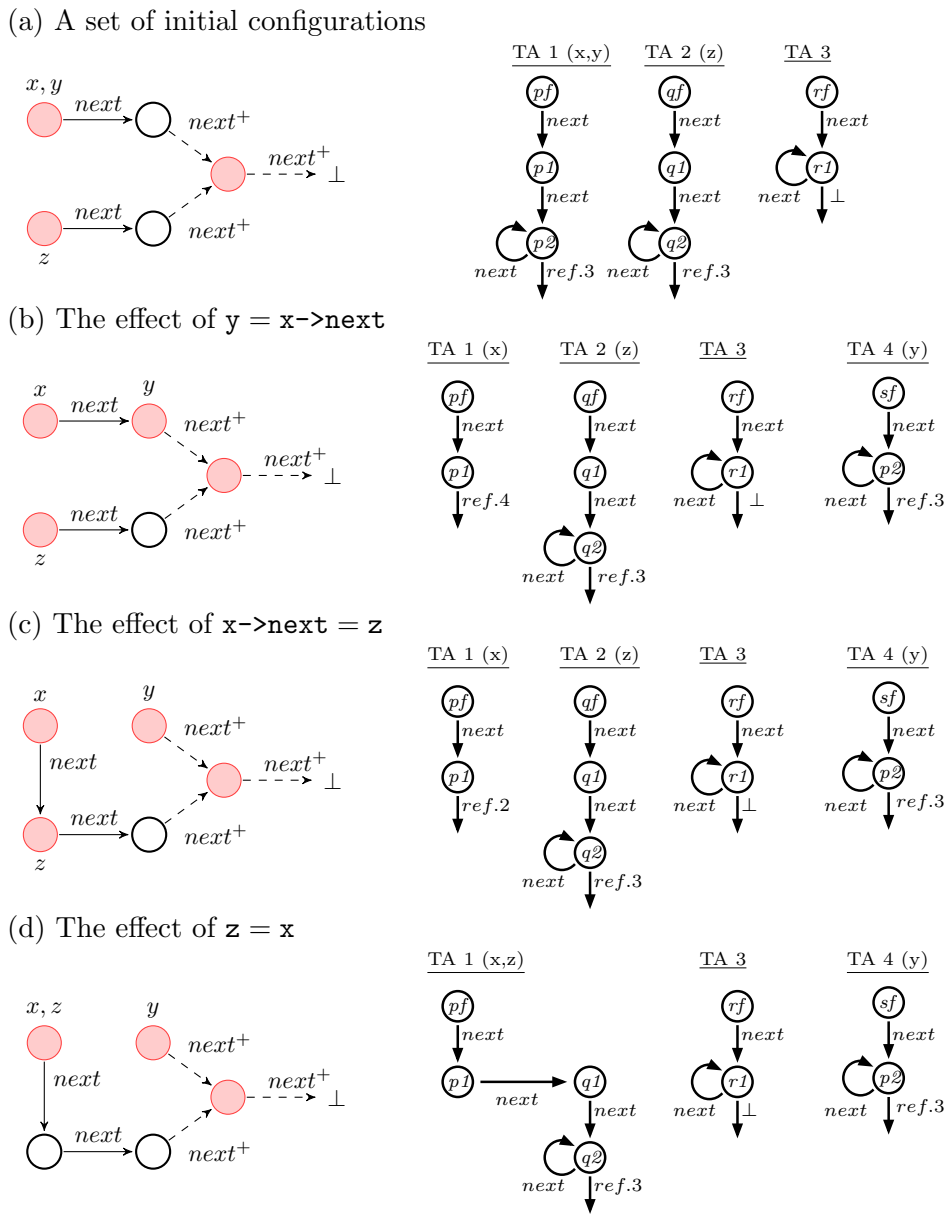


Figure 4.1: A concrete (on the left) and symbolic execution (on the right) of statements `y = x->next`, `x->next = z`, and `z = x`. For the sake of simplicity, the presented FA are not strictly in their canonical form.

to the newly created TA accepting the `s`-subtrees of the trees accepted by the original TA, the transition $a(\dots, q_s, \dots) \to q_f$ is substituted by the transitions $a(\dots, q_r, \dots) \to q_f$ and $r \to q_r$ where r is a reference to the TA accepting trees whose root is labelled by `y`.

- `x->s = d`: Each transition $a(\dots, q_{d'}, \dots) \to q_f$ of the TA accepting the nodes pointed by `x` where q_f is the accepting state and $q_{d'}$ corresponds to a node representing the current value of `x->data` ($d' \to q_{d'}$) is replaced by $a(\dots, q_d, \dots) \to q_f$ such that $d \to q_d$.

- `free(x)`: In the first phase, we split the TA that accepts nodes pointed by `x` to more TA such that all successors of the original root become roots as well (i.e., the modified TA represents a set of single memory nodes). Then, the TA accepting nodes pointed by `x` is removed from the given FA, and `x` is added to the label associated with the special `undef` node.

- Tests over pointer variables and data stored in cells pointed to by variables are evaluated by examining labels of the rules leading to the accepting states of the TA that accept trees whose roots represent nodes pointed by the concerned pointer variables. These rules contain all the needed information (e.g., testing equality of pointer variables means that these variables should be associated with the same root node). After each test, the original SFA is split into two SFA—one of them accepts the trees that satisfy the tested condition, and the other one accepts the trees which do not satisfy it.

The execution of the particular program statement is followed by checking whether all TA components within a newly obtained CFA are reachable from program variables. If an unreachable component is found we have encountered a memory leak.

Example of Symbolic Execution. A simplified example of a symbolic execution is provided in Figure 4.1. 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 representation (top-down tree automata are used within the example in order to improve the readability). The initial configuration is depicted in Figure 4.1 (a), and Figures 4.1 (b), (c), and (d) represent the sets of heaps obtained after successively applying the statements `x` = `y->next`, `x->next` = `z`, and `z` = `x`.

4.1.3 Introduction of Auxiliary Roots

In certain cases, one cannot execute the effect of a program statement directly on the FA at hand. For example, consider an FA $\mathcal{F}$ and the statement `y := x->s`. Here, for any hypergraph represented by $\mathcal{F}$, `x` points to a cut-point that corresponds to the roots of the trees accepted by some component TA of $\mathcal{F}$. We want `y` to point to the node which is reachable from the node pointed to by `x` via the selector `s`. After executing the statement, `y` will point to a cut-point. However, it may be the case that the node `x->s` (i.e., the node that is the successor of the node pointed to by `x` via the selector `s`) is currently not a cut-point, and it is accepted at an ordinary automaton state (not an accepting state as in the case of the root). Therefore, the TA accepting trees whose roots are pointed to by `x` has to be split into a new pair of TA such that the first automaton accepts trees that have a reference to the second automaton as the `s`-successors of their roots nodes, and the second automaton describes the part of the heap starting at the `x->s` nodes in the trees accepted by the original automaton (see Figures 4.1 (a) and (b)).

Note that it may happen that in some trees accepted by the given TA, a split is needed whereas in the others not. This can happen when in the trees accepted by the TA there is a tree where `x->s` is a root reference and another one where `x->s` is an intermediate node (accepted at an ordinary state). As an example, there may be sequences of `s`-selectors below the node pointed by `x` of length one or more. This can be represented by a TA with a single accepting state q, and the transition function

$$\Delta = \{\mathtt{s}(q) \to q, \mathtt{s}(r) \to q, \mathtt{null} \to r\}.$$

For length one, one does not need to introduce a new root since some root (`null` in this case) is already reached via `s`, which is, however, not the case for the other lengths. In such a scenario, the TA has to be first divided into two TA such that the first one accepts trees which need to be split whereas the other accepts trees which do not need such a split (this can be done by inspecting the transitions in the immediate neighbourhood of the accepting states), and then the split is done only in the latter case. In particular case of our example, we divide Δ into

$$\begin{array}{lcl} \Delta_1 & = & \{\mathtt{s}(r) \to q', \mathtt{null} \to r\} \\ \Delta_2 & = & \{\mathtt{s}(q) \to q', \mathtt{s}(q) \to q, \mathtt{s}(r) \to q, \mathtt{null} \to r\} \end{array}$$

where q' is the newly created accepting state.

We now formalise the problem of adding auxiliary roots under the assumption that the TA to be split contains a single transition leading to an accepting state only and that the accepting state does not appear inside any left hand side of a rule of the TA. A general TA can easily be transformed into a set of several TA such that they all satisfy these restrictions. Let $\mathcal{A} = (Q, \Sigma, \Delta, \{q_f\})$ be the TA that we want to split, let $a(\ldots, q_s, \ldots) \to q_f \in \Delta$ be the only transition that leads to q_f, and let $q_s \in Q$ be the state at which the nodes accessible via the selector `s` from the roots of the trees accepted by $\mathcal{A}$ are accepted. We replace $\mathcal{A}$ by TA $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\mathcal{A}_1$ references $\mathcal{A}_2$ via `s`. Formally, $\mathcal{A}_1 = (Q, \Sigma, \Delta', \{q_f\})$, $\Delta' = \Delta \setminus \{a(\ldots, q_s, \ldots) \to q_f\} \cup \{a(\ldots, q_r, \ldots) \to q_f, r \to q_r\}$ where r is a root reference to the newly created TA $\mathcal{A}_2$, and $\mathcal{A}_2 = (Q, \Sigma, \Delta, \{q_s\})$. Since this transformation may cause that many states of $\mathcal{A}_1$ and $\mathcal{A}_2$ become useless, the automata are subsequently reduced (by first removing useless states and subsequently by using, for instance, techniques for simulation-based reduction of nondeterministic automata).

4.1.4 Restricted Pointer Arithmetic

In addition to the statements listed at the beginning of Section 4.1, we also handle a restricted pointer arithmetic arising from the use of the `&` C operator (an operator taking the address of an object in memory) and the operators `+`/`-` defined over pointers. Note that a support of this feature is crucial, e.g., for dealing with linked data structures in the form used in system software such as the Linux kernel. There, one can, for instance, encounter data structures which are linked via embedded headers located at certain non-zero offset from

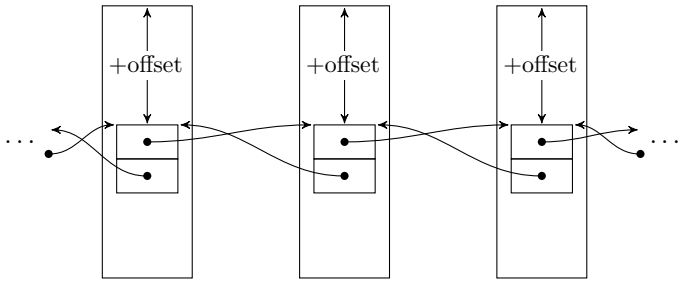


Figure 4.2: Nodes within a doubly-linked list linked via embedded headers

the beginning of allocated nodes (see Figure 4.2). When such a list is traversed, the cursor pointer going through the list follows the embedded headers, and pointer arithmetic is used to get to the actual list nodes.

Our support of basic pointer arithmetic is implemented as follows. First, we create a mapping $\mathit{offset} : \mathit{Sel} \to \mathbb{N}_0$ which translates each selector to a number representing the offset (in bytes) of that selector from the beginning of a structure allocated in memory. Furthermore, we associate selectors from Sel that label transitions of TA components of FA with integer *offset* values. Such an offset says that the pointer given by the concerned selector is not pointing to the node from which the concerned TA transition is executed but some number of bytes before or behind the beginning of that node in memory. Likewise, we associate integer offset values with pointer variables referring to the roots of FA. The execution of the statement `x = &y->s` is then performed by setting the content of the variable `x` to the content of the variable `y` and adjusting the offset associated with `x` to $\mathit{offset}(s)$.

The handling of `x = &y` is a bit more complicated as `&y` cannot be evaluated to a valid pointer within the proposed representation (as `y` is stored in the stack not the heap). In order to solve this problem, we have slightly extended our original model. In addition to representing the heap itself, the extended model also contains a special purpose singly-linked list representing the call stack. In this list, each active function call is represented by a single element holding all local variables of the corresponding function.

As a side-effect of this extension, one can even consider verification of simple recursive functions. This is achieved by applying abstraction to the list representing the call stack which allows one to obtain a configuration which represents an unbounded call chain. Note that since the function calls can be nested in an arbitrary way and each function declares different local variables, the nodes within the call stack do not share the same layout. However, in case of recursion, a single function is eventually called for the second time. This can be exploited by our abstraction that may widen the list representing the call stack from containing one occurrence of a loop in the recursive calls to any number of occurrences of the loop. Unfortunately, this approach is rather limited because the resulting data structure (including the stack) tends to be very complicated (often outside of the class which we can currently handle).

Algorithm 1: The Main Verification Loop

Input: a CFG P representing the program to be verified
Output: **safe** if P is safe, **unsafe** if P might contain an error

```
1  Visited := ∅;
2  Next := {(p_init, F_empty)};
3  while Next ≠ ∅ do
4      pick (p, F) ∈ Next;
5      Next := Next \ {(p, F)};
6      Visited := Visited ∪ {(p, F)};
7      foreach (a, q) ∈ successors(P, p) do
8          S' := unfold({F}, a);
9          S' := add-auxiliary-roots(S', a);
10         if violates-safety(S', a) then
11             return unsafe;
12         S' := execute(S', a);
13         if contains-garbage(S') then
14             return unsafe;
15         foreach F' ∈ S' do
16             F' := fold-normalise-abstract(F');
17             if L({F'}) ⊈ L({F'' : (q, F'') ∈ Visited}) then
18                 Next := Next ∪ {(q, F')}
19 return safe;
```

4.2 The Main Verification Loop

As we have briefly described in the introduction of this chapter and also in Section 4.1.2, our verification procedure performs a classical (forward) control-flow fixpoint computation over the CFG of the analysed program. The pseudo-code of this procedure is given as Algorithm 1 which we are now going to describe. In particular, Algorithm 1 represents the program configuration as a pair containing a location of the CFG and an FA representing a heap configuration together with values of program variables. Two sets of such configurations are maintained. $Visited$ contains all program configurations which have already been analysed. $Next$ contains those program configurations which still need to be processed.

The algorithm starts by populating $Next$ with a pair $(p_{init}, \mathcal{F}_{empty})$ consisting of the initial CFG location (denoted by p_{init}) and an FA representing the empty heap (denoted by $\mathcal{F}_{empty}$). The main loop iteratively picks program configurations $(p, \mathcal{F})$ from $Next$ as long as it is nonempty. The chosen configuration is then removed from $Next$, and it is added to $Visited$. In the next step, the algorithm enumerates all successors (a, q) of p (i.e., all pairs (a, p) for which there is an edge from p to q labelled by a) within the given CFG. For each (a, q), the algorithm proceeds as follows. First, if need be, the part of F that is accessed by a is unfolded on line 8 as briefly discussed in Section 4.1.2 and also in Section 4.3. This operation can in general yield an SCFA. Then, auxiliary roots needed for the execution of a are introduced on line 9 as described in Section 4.1.3 (which can again produce an SCFA even from a single FA on input). Next, the algo-

Algorithm 2: `fold-normalize-abstract`

Input: an FA $\mathcal{F}$
Output: an FA representing an abstraction of $\mathcal{F}$

```
1 repeat
2     F' := F;
3     F := fold(F);
4     F := normalise(F);
5     F := abstract(F);
6 until F = F';
7 return F;
```

rithm checks whether it is safe to execute a in the configurations represented by $\mathcal{S}'$ (i.e., there is no risk of causing null/undefined dereferences, etc.). In case it is not safe to execute a, the procedure immediately returns **unsafe**. Otherwise, a is symbolically executed on line 12 which can again transform a single FA into an SCFA. After the execution of a, the algorithm checks whether some of the generated FA contains components unreachable from program variables in which case the procedure again returns **unsafe** (a memory leak is detected). Next, the algorithm processes element-wise the SCFA $\mathcal{S}$, which represents the set of program configurations obtained from F by executing a. Each $\mathcal{F}' \in \mathcal{S}'$ is first iteratively folded, normalised, and abstracted on line 16 (see Algorithm 2 which will be described later). Then, the algorithm tests whether the newly obtained FA $\mathcal{F}$ is already covered by the program configurations reached at the given line previously (see Section 3.3.4). If it is not, the pair $(q, \mathcal{F}')$ is added to $Next$ on line 18.

If $Next$ becomes empty, the algorithm reports that the program is safe. The answer **unsafe** indicates that one needs to analyse the error trace (which is not covered by Algorithm 1). In order to detect spurious counterexamples and to refine the abstraction, one can use a *backward symbolic execution* similarly as in [BHRV06b] (see Section 2.5 for a brief description). Although we have not yet implemented the backward symbolic execution in our framework, we discuss the theoretical issues of executing a program backwards in Section 4.5.

We now get back to the process of folding, abstraction, and normalisation described in Algorithm 2 and discuss it in more detail. In the algorithm, the operation fold may heuristically select some parts of its input FA and fold them into nested FA (thus introducing a hierarchical encoding of the original FA) as discussed in Section 4.3. The operation *normalise* performs normalisation as described in Section 3.3.2 and Section 3.5.7, respectively. Finally, the operation *abstract* performs the actual abstraction by applying—to the individual TA inside the input FA—a specialisation of the general-purpose techniques described in the framework of abstract regular tree model checking [BHRV06b]. To recall, the abstraction collapses automata states with similar languages (based on their languages up-to some tree depth or using predicate languages). More details on the abstraction can be found in Section 4.4. The described operations are then repeated due to a need to handle the possibly hierarchically nested FA

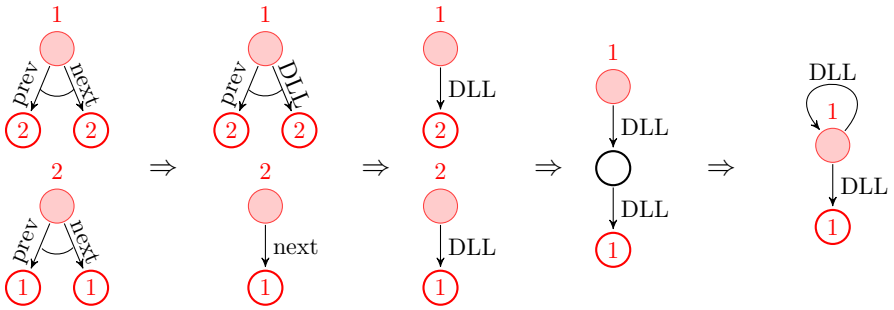


Figure 4.3: An example of folding, normalisation, and abstraction

(intuitively, these are iteratively folded and abstracted from their more nested levels to the top level).

To demonstrate how Algorithm 2 works, one can consider the FA representing two elements of a cyclic doubly-linked list depicted in Figure 4.3. First, the pairs of `next` and `prev` links become encoded hierarchically after the execution of line 3 of the algorithm. Then, the components are concatenated and reordered on line 4. This way, we obtain a single TA which represents both elements of the list. Finally, on line 5, the abstraction can transform the TA representing 2 elements of a cyclic doubly-linked list into a set of an arbitrary cyclic doubly-linked lists containing at least one element. If the cyclic doubly-linked list is a part of some more complicated structure (e.g., DLLs of cyclic DLLs), it might be necessary to repeat the whole process again. For this reason, we repeat the loop as long as something changes.

4.3 Folding of Nested Forest Automata

Originally, our verification approach as published in [HHR+11a] relied on the fact that:

1. the user is able to provide a set of nested SFA which is sufficient for the verification of the given program, and

2. it is enough to look for instances of the provided nested SFA (boxes) to be folded at roots existing in the FA in which the folding should occur.

However, in our experiments, it turned out that constructing a set of boxes suitable for verification of a given program may be quite complicated even for an experienced user. Moreover, in our implementation, it was not possible to reuse boxes constructed even for quite common sub-heap patterns—such as doubly-linked list segments—in cases where the structures were connected via different selectors. Finally, in certain cases, the algorithm could either eliminate an existing root by concatenating two components together, or it could also perform a folding of some box at this root. If the concatenation was performed first (during normalisation), it could happen that the box (whose folding was crucial for eliminating other cut-points which cannot be eliminated in any other way) could not be folded anymore because the required root disappeared, and

as a consequence, the tool did not terminate. Therefore, we have developed a fully automatic approach which is able to automatically find a suitable set of boxes and it also does not rely on the folding at existing roots only.

Recall that boxes have been introduced to bring a possibility of hiding certain cut-points which appear repeatedly within the heap. The way boxes suitable for dealing with different structures look like heavily dependents on the shape of the data structures being handled. However, it is usually the case that it is suitable to hide some infinitely occurring (possibly hierarchical) heap patterns. Looking for such patterns is what we concentrate on in the following.

The repeated sub-heap patterns to be used as boxes are to be sought in the CFA generated by the symbolic execution. A problem is that it is easy to encounter sub-heap patterns which are created during the execution of the program, they repeat a few times, but exist only temporarily. Unfortunately, folding these patterns into boxes can cause that the verification does not terminate (more precisely, folding a wrong pattern often triggers subsequent folding of other wrong patterns, and, as a result, such misfolding then often continues forever).

To overcome this issue, we tackle the problem from a different point of view. Instead of trying to discover repeating sub-heap patterns whose folding into boxes is suitable for dealing with the data structures generated by the program being verified (whose shape is unknown to us), we concentrate directly on dealing with the growing number of cut-points. This means that instead of the question "What repeated patterns does one need to fold?", we ask "Which cut-points need to be eliminated?". In what follows, we show how we can automatically eliminate certain kinds of cut-points using box folding.

4.3.1 Cut-point Types

We are now going to describe four types of cut-points (illustrated in Figure 4.4), which we distinguish for the purpose of cut-point elimination. These four types of cut-points follow from the way cut-points may arise. Namely, a cut-point arises when there is a loop in the heap on some node, or when some node is referenced multiple times. Moreover, both of these scenarios can happen either within a single component or across several different components, thus giving four ways how a cut-point may arise. Note, however, that the ways cut-points arise can be combined, meaning that a single cut-point may fall under several of the types at the same time.

In particular, Type 1 cut-points arise, for instance, when one works with cyclic lists. In such a case, a single TA component encodes a set of cyclic lists with the cycle encoded using a leaf which refers back to the root of the component. Type 2 cut-points are those cut-points which are referred multiple times from within a single component. These typically arise when one deals with trees whose all leaves refer to some designated node (e.g., the root). Next, a set of two or more cut-points is said to consist of cut-points of Type 3 if the components rooted at them are linked into a cycle. A typical scenario where such cut-points appear is working with doubly-linked lists where the cycle appears between a pair of successive list nodes. Observe, that each inner

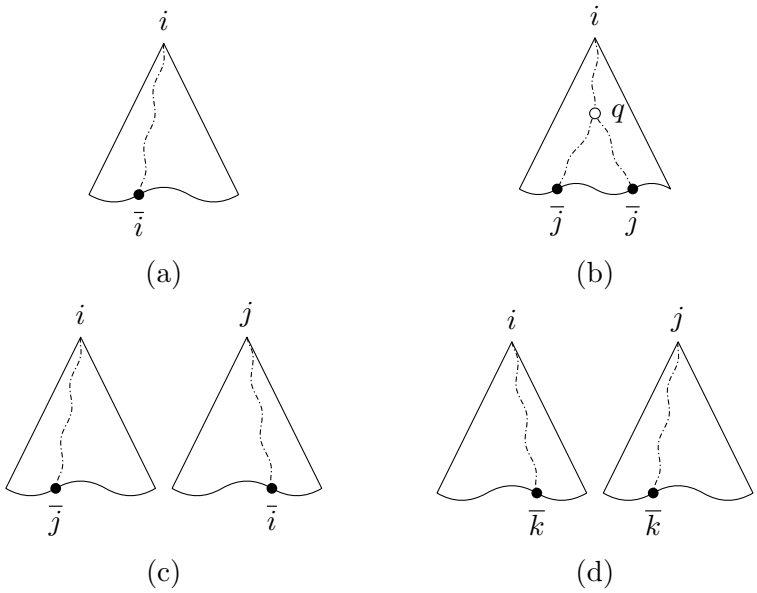


Figure 4.4: Four possible types of cut-points. (a) A Type 1 cut-point appearing as a reference in the same component whose root it is. (b) A single component containing multiple references to a Type 2 cut-point. (c) Two components mutually linked via Type 3 cut-points. (d) A Type 4 cut-point being referred from two other components simultaneously.

element of the lists is pointed from its predecessor and from its successor at the same time, and so (without using a hierarchical encoding) every element has to reside in its own component (and the present loops cannot be reduced to loops within a single component with a cut-point of Type 1). Finally, Type 4 cut-points are the cut-points referred from two or more components at the same time.

In the following section, we will show how to eliminate cut-points of Type 1 and 2 as well as some cut-points of Type 3. In the latter case, we restrict to dealing with pairs of cut-points of Type 3 that are the roots of neighbouring components linked in a cyclic way. We have not tried to go for the more general case yet since we have not encountered loops going through more than 2 cut-points in any of the considered case studies. On the other hand, cut-points of Type 4, for which we do not have any direct elimination procedure either, are quite ubiquitous. However, in all the case studies that we have considered, cut-points of Type 4 were cut-points of other types too. Moreover, it turned out to be sufficient to eliminate the parts of the heap causing these cut-points to be of Type 1, 2, or 3, and they stopped being cut-points of Type 4 too. For instance, in the case of doubly-linked lists, each node in fact corresponds to a cut-point of Type 3 and Type 4 at the same time. Nevertheless, when we fold the parts of the heap that make the concerned nodes to be cut-points of Type 3, there will not remain any cut-points of Type 4 either (see Figure 4.5 for an illustration).

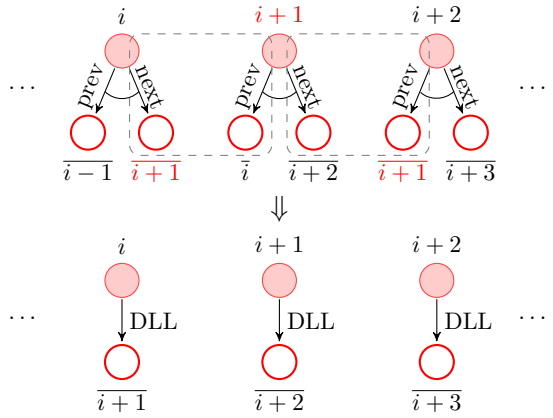


Figure 4.5: Folding of a doubly-linked list with a Type 4 cut-point highlighted in red

4.3.2 Component Slicing

Before we explain, how to deal with the particular types of cut-points, we first introduce the so-called *component slicing*. To perform folding on a single hypergraph, we first choose a sub-graph (sub-hypergraph) of this hypegraph which should be replaced by a hyperedge. Then, we remove the selected sub-graph and we add the hyperedge which then represents the removed substructure. We, however, perform folding not on a single hypergraph, but on a set of hypergraphs represented by some FA $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$. Therefore, whenever we want to perform folding of some part of a component $\mathcal{A}_i$ (or, in general, several components) into a box B, we need to proceed as follows. First, we identify a transition t of $\mathcal{A}_i$ at which we want to perform the folding (or, we identify a transition in each component in which the folding should occur at the same time), and we decide which part of t will be hidden inside the (not yet known) box B. This way, we choose which sub-graphs will be hidden—in particular, we will hide the selected part of the transition and everything what is (top-down) reachable from that part of the transition in $\mathcal{A}_i$.

In order to obtain B, we split t into two new transitions t_k and t_r such that t_k is the part of t to be put into B and t_r contains the part of t not contained in t_k. Further, we create two new components (automata) called the *kernel* and the *residue*. The kernel contains t_k as an accepting transition (i.e., a transition with an accepting state on its right-hand-side) together with all other transitions (top-down) reachable from t_k within $\mathcal{A}_i$. As a result, the kernel defines what B is. The residue is obtained by replacing t by t_r in $\mathcal{A}_i$. If we now directly replaced $\mathcal{A}_i$ by the residue in $\mathcal{F}$, we would obtain a set of hypergraphs in which the selected sub-graphs are entirely removed. To simulate the replacement of the part of the hypergraph by the hyperedge, we append B to t_r inside the residue. The process of splitting a selected component into kernel and residue is called component slicing (see Figure 4.6 for an illustration of this concept).

To describe component slicing more precisely, we now recall the automata representation described in Section 4.1.2. Essentially, we work with transitions labelled by structured alphabet symbols eac of which encodes a sequence of se-

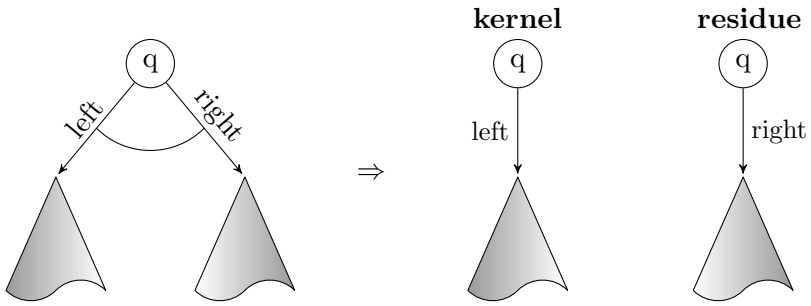


Figure 4.6: An example of component slicing. The left part shows the original automaton which is to be sliced at the state q according to the edge labelled by "left" (i.e., we perform $(\langle \mathit{left}, \mathit{right} \rangle(q_1, q_2) \to q) \rhd \{\mathit{left}\}$). The right part shows the result after slicing in which the kernel contains the structure reachable via "left", and the residue contains the rest of the original automaton (in this case, the structure reachable via "right").

lectors (or, in general, a sequence of selectors or boxes) such that there is a fixed correspondence between the selectors (boxes) that appear within an alphabet symbol and the (ordered) tuple of predecessor states of the transition labelled by that symbol. For instance, we can consider a transition $a_1 a_2(q_1, q_2) \to q$ such that $\#_e(a_1) = \#_e(a_2) = 1$ where a_1 leads to the first predecessor state (q_1), and a_2 leads to the second predecessor state (q_2). For transition splitting, we introduce an operator called *transition cut* operator ($\rhd$) which when given a transition $t = \alpha(q_1, \ldots, q_n) \to q$ and a set of selectors (boxes) E produces a new transition t' such that t' contains only the parts of the symbol α included in E together with the predecessor states corresponding to these parts.

In order to give a formal description of the operator $\rhd$, we need to introduce several auxiliary operations. In particular, let $t_1 = (q_1, \ldots, q_n)$ and $t_2 = (r_1, \ldots, r_m)$ be two tuples (of states). We define the concatenation of t_1 and t_2 denoted by $t_1 \circ t_2$ as the $(n + m)$ tuple $(q_1, \ldots, q_n, r_1, \ldots, r_m)$. We also define a tuple selection operation denoted by $t_1[i, k]$ which produces the k-tuple $(q_i, \ldots, q_{i+k-1})$ provided that $1 \leq i \leq n$ and $i + k - 1 \leq n$. To summarise:

$$\begin{array}{lcl} (q_1, \ldots, q_n) \circ (r_1, \ldots, r_m) & = & (q_1, \ldots, q_n, r_1, \ldots, r_m) \\ (q_1, \ldots, q_n)[i, k] & = & (q_i, \ldots, q_{i+k-1}). \end{array}$$

We aim at breaking a transition $t = a_1 \ldots a_n(q_1, \ldots, q_m) \to q$ into pieces composed of a_i and the predecessors connected to a_i. Therefore, we further define a function called $\mathit{pick}(a_1 \ldots a_n(q_1, \ldots, q_m) \to q, i)$ which selects the tuple of predecessor states corresponding to a_i from the transition $a_1 \ldots a_n(q_1, \ldots, q_m) \to q$ (provided $1 \leq i \leq n$ and $\#_v(a_1 \ldots a_n) = m$):

$$\mathit{pick}(a_1 \ldots a_n(q_1, \ldots, q_m) \to q, i) = (q_1, \ldots, q_m)[\#_v(a_1 \ldots a_{i-1}) + 1, \#_e(a_i)].$$

Using the previous definition, we define a function called *split* which splits a transition into a tuple of pairs containing parts of the original symbol and the connected tuples of predecessor states:

$$\mathit{split}(t = a_1 \ldots a_n(q_1, \ldots, q_m) \to q) = ((a_1, \mathit{pick}(t, 1)), \ldots, (a_n, \mathit{pick}(t, n))).$$

In order to transform a tuple containing separated parts of the symbol and corresponding tuples of states into a transition, we define a complementary function to *split* called *join* which, given a tuple of pairs of the above form and a state, creates a new transition as follows ($(\mathbf{s_1} \dots \mathbf{s_n})$ denotes the concatenation of tuples $\mathbf{s_1} \circ \mathbf{s_2} \circ \dots \circ \mathbf{s_n}$):

$$join(((a_1, \mathbf{s_1}), \dots, (a_n, \mathbf{s_n})), q) = a_1 \dots a_n(\mathbf{s_1} \dots \mathbf{s_n}) \to q$$

At this point, we are able to split a transition into pieces and then join these pieces back together. We, however, do not intend to join all of the original pieces but only some subset of them. Therefore, we also define a filtering function Φ_f, which (depending on a Boolean function f) removes certain elements of the tuple it receives as the input:

$$\begin{array}{lcl} \Phi_f() & = & () \\ \Phi_f(x_1, \dots, x_n) & = & \begin{cases} (x_1) \cdot \Phi_f(x_2, \dots, x_n) & \text{if } f(x_1) \\ \Phi_f(x_2, \dots, x_n) & \text{otherwise} \end{cases} \end{array}$$

Finally, we are ready to give the definition of the operator $t \rhd E$ which first splits the given transition t, then it keeps only those pairs containing elements of the set E, and, in the last step, it joins the remaining pairs into a new transition:

$$(t = \alpha(q_1, \dots, q_n) \to q) \rhd E = join(\Phi_f(split(t)), q)$$

where $f((a, \mathbf{s})) \iff a \in E$.

Now, for a given state q, a fixed symbol $\alpha = a_1 \dots a_n$, and a given set of selectors (boxes) $E \subseteq \{a_1, \dots, a_n\}$, slicing of a component TA works as follows. We first create a kernel and a residue as two separate copies of the component to be sliced. Then, in the kernel, we replace each transition t of the form $\alpha(q_1, \dots, q_m) \to q$ by the transition $t \rhd E$. Contrary to that, we replace each transition t of the form $\alpha(q_1, \dots, q_m) \to q$ by the transition $t \rhd (\{a_1, \dots, a_n\} \setminus E)$ in the residue.

Note that all steps within our symbolic execution produce automata in which all transitions of the form $\alpha(\dots) \to q$ are labelled by the same symbol α (i.e., $\{\alpha(\dots) \to q, \alpha'(\dots) \to q\} \subseteq \Delta \Rightarrow \alpha = \alpha'$). Moreover, the abstraction never collapses states q_1 and q_2 if there exist transitions $\alpha(\dots) \to q_1$ and $\alpha'(\dots) \to q_2$ such that $\alpha \neq \alpha'$ (see Section 4.4). Hence, we assume that the symbol is always specified implicitly by the state q, and we in the following parametrise component slicing by a state and a set of selectors (boxes) only.

4.3.3 Cut-point Elimination

In this section, we discuss the automatic elimination of cut-points of type 1, 2, and 3. As we have already mentioned, it might happen that some cut-point is of more than one type which is solved by several applications of the folding algorithm. The heuristic, which currently seems to give the best results in our experiments, eliminates cut-points in the order 3, 2, and 1.

Type 1. A Type 1 cut-point (c.f. Figure 4.4 (a)) corresponds to a component which contains references to its own root (these are called self-references in the following). To eliminate such cut-point, we want to "hide" all the self-references into a box. Therefore, we identify a set E of all selectors (or boxes) within the accepting transitions which lie on some path going to some self-reference (for this, we conveniently use labels introduced in Section 3.3.2). Then, we perform slicing of the component parametrised by the appropriate accepting state and the set E from the previous step to obtain a kernel containing all self-references and a residue containing the rest. Next, we perform a DFT on the kernel and we rename all references $R \subseteq \{1, \ldots, n\}$ that appear in the kernel according to the order in which they are visited (such that the self-references are always relabelled to 0) to obtain a mapping $f : R \to \{0, \ldots, |R| - 1\}$. The relabelled kernel is transformed into a new box B such that $\#_e(B) = |R|-1$. All accepting transitions $\alpha(q_1, \ldots, q_n) \to q$ of the residue are modified to

$$\alpha B(q_1, \ldots, q_n, r_1, \ldots, r_{\#_e(B)}) \to q$$

where r_j is a state representing a reference to a root u such that $f(u) = j$ (i.e., the additional predecessor states encode the mapping f and, as a result also the correspondence between the root references appearing inside the box and the root references appearing on the level on which the box is used)[4]. As the last step, we replace the original component by the modified residue.

Note that the process of folding can easily be reversed whenever needed. First, we extract the mapping f. Then, we relabel the root references inside the box using f^{-1}, and we replace the given box in some transition t by the relabelled component.

Type 2. A Type 2 cut-point arises when the reference to that cut-point appears multiple times within a single component (c.f. Figure 4.4 (b)). In this case, we are therefore interested in folding the smallest part of the component containing all the references inside a box which will then allow us to reduce the number of references to the given cut-point to one whenever used. In order to perform the folding as efficiently as possible, we first identify a state q whose subtrees contain all the root references to the given cut-point and none of its predecessor states has this property (see Figure 4.4 (b)). If there are more such states, the folding is performed separately for each of them. In the next step, we identify the set of selectors (boxes) E lying on some path to the given cut-point reference starting at a node corresponding to q. Then, we perform slicing of the component parametrised by q and E. The kernel of the slicing is transformed into a box B which is then appended to the residue in the same manner as in the case of cut-points of Type 1.

Type 3. To eliminate a pair of Type 3 cut-points i,j (i.e., the cut-points mutually referring to each other—see Figure 4.4 (c)), we need to use a box encoding

[4]Here, for the sake of simplicity, we ignore the fact that the selectors and the boxes are ordered, hence B should not be put simply behind α. However, the procedure can easily be extended to respect the ordering.

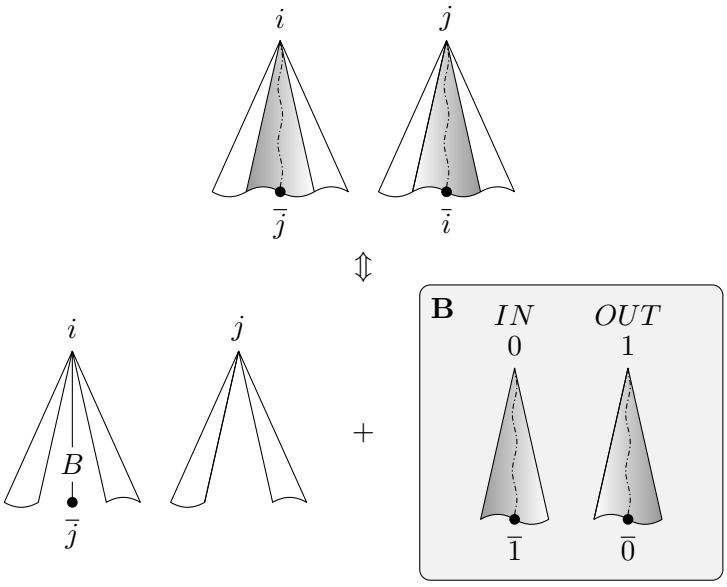


Figure 4.7: An illustration of an elimination of Type 3 cut-points. The two components are first sliced at their accepting states. The kernels of the result containing references to i and j are transformed into a new box which is added to the remaining part of the component i. At the end, the modified component j contains no references to i.

a cycle starting at cut-point i going through cut-point j and finally back to i (this corresponds, e.g., to a pair of forward and backward links in a doubly-linked list). Intuitively, we create a box containing two components (unlike cut-points of Type 1 and 2 which contain only a single component). The first component of the box will contain the part of the component corresponding to i and containing all references to j. Analogically, the second component of the box will contain the part of the component corresponding to j and containing all references to i. As a result, the box will represent a structure containing the loop i-j-i.

Let us describe the elimination of Type 3 cut-point in greater detail. First, we perform slicing of components i and j which mutually refer to each other such that the kernels contain all paths from i to j (or from j to i, respectively). The two kernels are transformed into a box B. The box B is then appended to the accepting transitions of the residue of the component i in the same way as in the previous cases. As the last step, we replace the component i within the original FA by the modified residue and the component j by the residue of component j. As a result, the new component i contains a single reference to j, and new component j contains no reference to i—see Figure 4.7.

Note that one could also perform the symmetrical transformation in which the newly created box is added into the component j. This would yield a different structure with the same semantics. In order to decide which variant to choose in practice, we use an ordering on the set of components of the original FA, that is if $i < j$ we append the newly created box to the residue of i.

4.3.4 From Nested FA to Alphabet Symbols

When performing certain automata operations such as language inclusion, we treat all symbols as not having any underlying structure at all. However, to make this work, we have to ensure that the nested FA (boxes) with same semantics are represented via the same alphabet symbol. This is achieved by maintaining a database which maps boxes to alphabet symbols. Every time a new box is created, it is first compared to existing boxes having the same root interconnection graph (which also implies the same number of components). The comparison of two boxes is done via checking language equality of FA using which they are represented[5][6]. If the same box already exists, the symbol obtained from the database is used for its representation. If it does not, then a new alphabet symbol is created and it is stored within the database.

Furthermore, we can also consider only language inclusion to relax the requirement of equality when testing whether the two boxes match. This means that during the folding the given box can be potentially replaced by a "bigger" one which already exists in the database. As discussed later, this can in some cases drastically decrease the number of boxes which are required and thus substantially increase the performance.

4.4 Abstraction

Due to the fact that we are dealing with heaps, the set of all reachable configurations at a given program location is usually infinite. These infinite sets can often be described finitely using a finite set of FA. However, a naïve iterative way of computing a set of FA describing the set of all reachable configurations will typically not terminate, producing an infinite sequence of different FA under-approximating the set of all reachable configurations. To make the computation terminate as often as possible, we apply abstraction as is usual in the framework of abstract regular tree model checking. We, in particular, build on the general-purpose finite height abstraction introduced in [BHRV06b] which we, however, significantly modify to cope well with the various special features of our automata. Namely, it has to work over tuples of TA and cope with the interconnection of the TA via root references, with the requirement of canonicity, with the hierarchical structuring, as well as with the fact that we generate *sets* of FA at each line of the program, not just individual FA.

In particular, first of all, we apply the TA abstraction to each component of FA separately. Further, we slightly modify the finite height abstraction by ignoring the accepting status of automata states. Next, we restrict the abstraction such that if it is applied on a CFA, then the result is also a CFA (i.e.,

[5] Here, we are not dealing with sets of FA, hence the comparison of two FA can be done component-wise.

[6] The language equality check is performed as two inclusion queries. The inclusion test itself is again only a safe approximation as we ignore the structure of nested boxes which can appear within the box we have just created. Therefore, it can be the case that two boxes with the same semantics are represented via different alphabet symbols. However, according to our experiments, such approximation works very well in practice.

the abstraction preserves the canonicity respecting status). Subsequently, we propose a further refinement of the abstraction based on more precise tracking of the paths between roots and root references even in cases when some of these paths are hidden in nested FA. Finally, we show how the abstract symbolic computation may be accelerated by enriching CFA that are generated at a certain line by the languages of the other CFA generated at the same line as a part of a single SCFA.

The rest of the section is organised as follows. We first introduce the principle of quotienting automata w.r.t. an equivalence relation as a basis for our abstraction. Next, we recall the equivalence based on languages of trees up to some height introduced as a general-purpose equivalence for abstracting automata in the framework of abstract regular tree model checking (which we slightly modify by ignoring the accepting status of states). Subsequently, we introduce our specialised abstraction which preserves canonicity, followed by combining it with the basic finite height abstraction. Afterwards, we propose a further refinement of the abstraction which allows one to distinguish (to some extent) trees having different sets of paths (possibly hidden in nested FA) between their roots and the particular root references. Finally, we show how one can abstract SCFA in a better way than by separately abstracting each of their CFA.

4.4.1 Automata Quotienting

We now formalise the notion of equivalence-based *automata quotienting*. Let $\mathcal{A} = (Q, \Sigma, \Delta, F)$ be a tree automaton and $\approx$ be an equivalence on Q. The *quotient automaton* $\mathcal{A}/\approx$ is obtained from $\mathcal{A}$ by collapsing its states according to $\approx$ (i.e., the set of states of $\mathcal{A}/\approx$ is obtained by taking the partition of Q induced by $\approx$). Let $[q] = \{q' \in Q : q' \approx q\}$. Then,

$$\mathcal{A}/\approx = (\{[q] : q \in Q\}, \Sigma, \Delta', \{[q] : q \in F\})$$

where $\Delta' = \{a([q_1], \ldots, [q_n]) \to [q] : a(q_1, \ldots, q_n) \to q \in \Delta\}$. We extend the previous notion to a forest automaton $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ by creating quotient (tree) automata for each component separately:

$$\mathcal{F}/\approx = (\mathcal{A}_1/\approx, \ldots, \mathcal{A}_n/\approx, R).$$

Next, we need to clarify what relation $\approx$ we use in order to obtain the abstraction. In fact, we use two relations over the set of states. Our abstraction is based on the *equivalence of languages of bounded height* (see [BHRV06b] or Section 2.5), but we also want to preserve the overall shape of the heap, and therefore we also use a structure-preserving equivalence based on the reachability of root references. We discuss both of the equivalences in the following subsections.

4.4.2 Equivalence of Languages of Bounded Height

Our equivalence of languages of bounded height is based on the idea of finite height abstraction from [BHRV06b], but instead of considering languages of

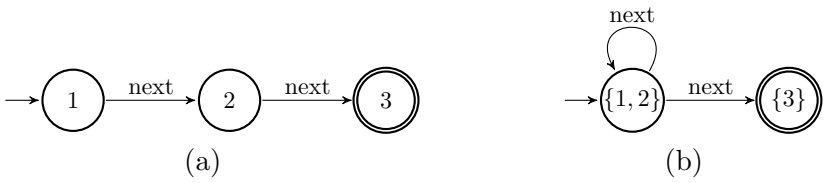


Figure 4.8: An example of abstraction. (a) An automaton representing a list containing 2 elements. (b) An abstracted automaton representing a set of lists containing at least 1 element obtained using $\approx_1$. For the sake of simplicity, we use *word* automata notation here.

trees of some restricted height, it considers sets of all trees of height at most k that can be read by the automata while getting to some state. This approach is, in fact, a generalisation of trace languages considered in [BHV04] for words to trees.

Formally, let $\mathcal{A} = (Q, \Sigma, \Delta, F)$ be a tree automaton and t be a tree over Σ. The *tree stub* of t of height k (denoted by $t_{|k}$) is the restriction of t defined as

$$t_{|k} = t|_{\bigcup_{i=0}^{k} \mathbb{N}^i} = \{(p, a) : p \in dom(t) \cap \bigcup_{i=0}^{k} \mathbb{N}^i \wedge t(p) = a\}.$$

Then, for a state q, the language of height k denoted by $L_{|k}(q)$ is the set of all tree stubs of height n of trees which can be accepted by q:

$$L_{|k}(q) = \{t_{|k} : t \in L(q)\}.$$

Finally, we say that the states q_1 and q_2 are equivalent up to height k iff they have the same languages of height k:

$$q_1 \approx_k q_2 \iff L_{|k}(q_1) = L_{|k}(q_2).$$

In practice, we have obtained the best results with using the simplest $\approx_1$ equivalence (refined by the below described specialised equivalences). The effect of the abstraction based on the $\approx_1$ equivalence is depicted in Figure 4.8. Using the $\approx_1$ equivalence means that we allow the abstraction to treat memory nodes with the same layout of selectors (or, roughly speaking, memory nodes of the same type in the typing system of the C language) as equal no matter what data they contain[7]. Using this information together with the below introduced specialised refinements turned out to be sufficient in all the considered case studies. On the other hand, attempts to replace some of the specialised refinements by increasing the abstraction height were unsuccessful due to either not being precise enough or due to preventing too many states from being collapsed and thus causing an explosion in the size of the automata being handled.

[7]Indeed, the values of data are represented by separate child nodes. In order to make the abstraction data-aware, one needs to use at least $\approx_2$.

4.4.3 Canonicity Preserving Equivalence

Given an FA $\mathcal{F}$, the abstraction based on the equivalence of languages of bounded height can in general produce some FA $\mathcal{F}/{\approx_k}$ which is not consistent, i.e., it might happen that one could pick a forest F from the corresponding components of $\mathcal{F}/{\approx_k}$ such that $\otimes F$ is not defined (one can, for example, obtain a forest in which one attempts to define the same selector multiple times for a single node). Moreover, we also want the abstraction to keep the basic shape of the heap unchanged, meaning that the abstraction should not touch the interconnection among the components. In other words, if a certain root is referenced only once before the abstraction, it should be referenced only once after the abstraction as well, no cut-point reference should suddenly appear or disappear, etc. In order to better understand why this is important, one can imagine having a tree which contains a single root reference to some particular node (represented as a root of a different component). Now, an unrestricted abstraction could create a new tree containing many references to that particular node which would essentially transform the tree into a directed acyclic graph. This behaviour is, however, not desired in a large majority of cases as it almost always causes that the analysis immediately fails by reaching a spurious counterexample. Additionally, we also require that the abstraction applied on a canonical FA always produces an FA which is also canonical. To be precise, we want to ensure the following:

1. The order of roots visited during the depth first traversal of FA remains unchanged.
2. If a given component contains exactly one reference to a certain root before the abstraction, it also contains exactly one reference to that root after the abstraction.
3. The backward reachability relation remains unchanged.

To satisfy these requirements, we will take advantage of the labelling introduced in Section 3.5.7. Therefore, we first compute the labelling function $\mathit{label}(q) = (w, Y, Z_1, Z_2)$ introduced in Section 3.5.7 for each state q which can easily be done assuming that we are working with canonicity respecting FA. Then, we define the so-called *canonicity preserving equivalence* $\approx_C$ using the labelling function label:

$$q_1 \approx_C q_2 \iff \mathit{label}(q_1) = \mathit{label}(q_2).$$

It is not difficult to see that conditions (1), (2), and (3) from above are satisfied under $\approx_C$ due to the fact that the equivalent states are required to have the same labelling. Indeed, respecting the string w guarantees that the collapsed states accept trees with identical order of root references on the frontier. Moreover, w and Y together guarantee that the second condition is satisfied as well. Finally, respecting Z_1 and Z_2 preserves the backward reachability relation.

In practice, as already indicated, we combine the two above introduced equivalences, using finite height abstraction $\approx_1$ based on the $\approx_1$ equivalence. This means that we use the $\approx_1 \cap \approx_C$ equivalence for collapsing states of the encountered automata (apart from some exceptions discussed in the following section).

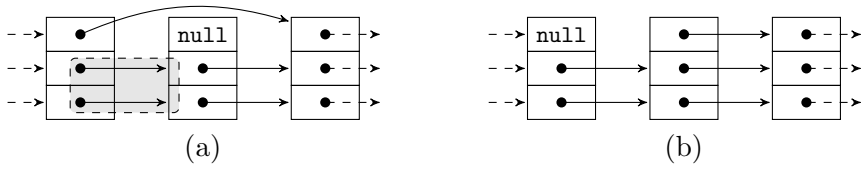


Figure 4.9: (a) A segment of a skip list in which the two links highlighted in grey will eventually become represented using a nested FA. (b) An incorrect over-approximation in which the top level list becomes disconnected.

4.4.4 Refined Canonicity Preserving Equivalence

During the experimental evaluation of our framework, we have found that the above abstraction fails for some of the more advanced examples (namely programs manipulating skip lists—see Section 4.6.4). An analysis of the spurious counterexamples revealed that in these cases, our abstraction is not able to properly distinguish two automata states if two trees from the languages of these states contain a different number of unique paths to some root reference, but some of these paths are hidden inside nested FA.

For instance, one can consider a skip list segment (see Section 4.6.4 for more details) depicted in Figure 4.9 (a). In this segment, the two links highlighted in grey will eventually become represented using a nested FA (before that, the first two nodes must reside in separate components). As soon as this happens, the abstraction based on $\approx_1 \cap \approx_C$ is not able to distinguish the first and the second node since it only considers that both have more than one path leading to the third node. As a result, FA encoding skip lists of some length may get abstracted such that the abstraction will include the structure depicted in Figure 4.9 (b), which is not a valid skip list because the top level layer is disconnected. Note that increasing the height of the abstraction, i.e., using $\approx_k$ for $k > 1$, will only defer the problem to longer skip list segments.

In order to prevent the above mentioned problem, one could think of extending the labelling introduced in Section 3.5.7 by an additional component recording the set (or at least the number) of distinct paths leading from nodes of trees to each root reference appearing in the trees. Unfortunately, this is not possible since the number of such paths is not bounded in general and, as a result, it is not guaranteed that the number of such labels would be finite.

However, it turns out that to avoid all spurious counterexamples arising during the verification of all the programs we considered it suffices to approximate the sets of paths leading from nodes of the trees to the different root references by recording the numbers of selectors through which one can (eventually) get to the particular root reference. This, for instance, helps to distinguish the first and the second node depicted in Figure 4.9 (the first node has 3 selectors through which one can get to the right-most node while the second node has only 2 such selectors).

Formally, let $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ be a hierarchical FA in which $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$ for $1 \leq i \leq n$. We label each $q \in Q_i$ by a set of labels (w, Y_1, Y_2, Z_1, Z_2) for which there is a tree $t \in L(q)$ such that:

- w, Z_1, Z_2 are as in Section 3.5.7.
- Y_1 is the same as Y in Section 3.5.7.
- $Y_2 \subseteq \{1, \ldots, n\} \times \mathbb{N}$ is a set such that $(r, k) \in Y_2$ iff the root reference r can be reached from the root of t via paths starting by k different selectors.

In order to compute the labelling, we again start by first labelling states w.r.t. the leaf rules, and then we propagate the so-far obtained labels bottom-up. The values of w, Y_1, Z_1, Z_2 are computed as before, and Y_2 is obtained as follows. For a fixed reference r and a transition $a_1 \ldots a_n(q_1, \ldots, q_m) \to q$ such that $\#_v(a_1, \ldots a_n) = m$, we compute

$$k = \sum_{i=1}^{n} \sum_{j=1}^{\#_e(a_i)} \mathit{leadsto}(\#_v(a_1 \ldots a_{i-1}) + j) * \mathit{selectors}(a_i, j)$$

where $\mathit{leadsto}(i) = 1$ iff there exists $k' > 0$ such that $(r, k') \in Y_2(q_i)$ or $\mathit{leadsto}(i) = 0$ otherwise, and $\mathit{selectors}(a_i, j) = l$ iff the output port j of a_i can be reached from the input port of a_i via paths starting by l different selectors (or $\mathit{selectors}(a_i, j) = 1$ iff a_i corresponds to a plain selector). Then, we add (r, k) to $Y_2(q)$. To obtain the number of different selectors that can appear at the beginning of the paths leading between the input and the output port of some nested FA, we use the Y_2 labelling computed for the nested FA itself.

Next, we compute the labelling function $\mathit{label}(q) = (w, Y_1, Y_2, Z_1, Z_2)$ for each state q in a way mentioned above, and we define the *refined canonicity preserving equivalence* $\approx_{C\uparrow}$ using the labelling function label:

$$q_1 \approx_{C\uparrow} q_2 \iff \mathit{label}(q_1) = \mathit{label}(q_2).$$

It immediately follows that two states can be collapsed using $\approx_{C\uparrow}$ iff they can be collapsed using $\approx_C$ and, moreover, the corresponding trees have the same number of selectors that can appear at the beginning of the paths leading from the root to each of the different root references.

According to our experiments, the abstraction based on $\approx_1 \cap \approx_{C\uparrow}$ is sufficient for the verification of all of the considered examples. However, if the abstraction generates some counterexample in the future (which is for sure going to happen, e.g., when one tries to enrich the technique for dealing with data-dependent data structures, such as red-black trees, or data dependent properties, such as sortedness), it is possible to either adjust (increase) the height of the abstraction or to further refine the abstraction using *predicate languages* (see again [BHRV06a] or Section 2.5). Moreover, it would be also interesting to see whether the predicate language abstraction could find the proposed refinement of the canonicity preserving equivalence automatically (even if this would cause some performance hit).

A proper implementation of a fully automatic abstraction refinement based on the predicate language abstraction remains an interesting subject for future research. In Section 4.5, we will, however, make the first step towards this goal by proposing a way how the verified program can be executed backwards along a suspected counterexample trace, which is a necessary precondition for predicate language abstraction.

```
struct Tree* buildTree() {
1: struct Tree* root = NULL;
2: while (<cond>) {
3:   struct Tree** n = &root;
4:   while (*n != NULL)
5:     n = &(<cond>?(*n)->left:(*n)->right);
6:   *n = newNode();
7: }
8: return root;
}
```

Figure 4.10: A simple C function which constructs arbitrary binary trees. We assume that `Tree` is a C structure containing `left` and `right` pointer fields (selectors). `<cond>` represents a condition which is abstracted away. `newNode` is a function which creates a new node and returns its address.

4.4.5 Abstraction for Sets of FA

According to our experiments, the approach presented above works quite well for many list-like data structures. One can observe that it is often sufficient to build a linked list containing 2 elements (assuming that one uses the height-1 language abstraction) which is then abstracted to the set of lists containing one or more cells. This usually amounts to symbolically executing 2 iterations of each loop within the program, which can be computed quite quickly. However, when one starts to deal with trees, the number of nodes, which has to be constructed before the abstraction collapses enough states, grows considerably, especially for trees of higher arities.

For instance, one can consider the function depicted in Figure 4.10. Here, the statements on lines 2–7 are iteratively executed during the verification.

Whenever the symbolic execution reaches a conditional statement or a loop statement (as, e.g., on lines 2 and 5), and the involved condition does not evaluate to either true or false in the given symbolic configuration, the configuration is split into two configurations in which the condition either holds or does not hold (as described in Section 4.1.2). The symbolic execution then continues into both of the branches with one of the symbolic configurations arising from the split processed right away and the other one postponed. Now assume that the verification algorithm always first explores the left branch on line 5. Therefore, configurations representing trees consisting solely of a left branch of an increasing height are first generated. The generation of such trees of an increasing height continues until the abstraction applies on line 4, producing a set of trees consisting of an arbitrarily long left branch. At this point, the execution branch is not explored any further as it does not produce new configurations. Then, the algorithm switches back to some of the postponed branches, in which one right successor is generated. After that, one starts to explore the left-most branch again, which eventually yields a set of trees in which the root has the

left successor or both successors, but those have left successors only. Eventually, the algorithm inspects a branch which at the end yields the fixpoint of the while-loop in the form of a set of arbitrary binary trees (see Figure 4.11 for an illustration of the possible evolution of the set of reachable configurations in the considered program). However, it is not difficult to see that this approach is far from optimal. Indeed, one would like to first explore the branch yielding the desired fixpoint. Then, while switching back the postponed branches, the verification procedure could quickly detect that no new configurations can be generated. Unfortunately, it is not possible to efficiently decide beforehand which branch to take in order to reach the fixpoint sooner.

In order to avoid the above problem, it would help to have a single CFA representing all configurations reached at the given program location through all explored execution branches. A symbolic execution from such a CFA would then correspond to executing all the branches at the same time. Such a CFA, however, cannot be obtained since CFA are not closed under union even if they have the same number of components and the same root interconnection graph (see Section 3.3.3).

Nevertheless, despite the fact that we are forced to work with SCFA, we will now show that it is possible to use the information contained at least in all the CFA that have the same number of components and the same interconnection graph to improve the performance of the abstraction, especially in cases with many execution branches (as above). This can be done as follows. Let $\mathcal{S}$ be an SCFA for the given program location. To compute an abstraction of $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R) \in \mathcal{S}$, we first select the subset $\mathcal{S}'$ of all the CFA of $\mathcal{S}$ that have n components and the specification R. Then, we unite the sets of states and transitions of the corresponding components of CFA in $\mathcal{S}'$ (assuming w.l.o.g. that these sets are disjoint) to obtain $\mathcal{F}'$ which itself inherits the set of final states from $\mathcal{F}$. As a result, the languages of $\mathcal{F}$ and $\mathcal{F}'$ are equal, but $\mathcal{F}'$ can also contain states and transitions from the CFA in $\mathcal{S}'$ which may have been created in other execution branches than $\mathcal{F}$. These additional states and transitions do not lead to any any final state, but when $\mathcal{F}'$ is abstracted, it can happen that some transitions which are unreachable in $\mathcal{F}'$ can become reachable in $\mathcal{F}'/{\approx}$. This in turn means that the resulting language will be enriched by some information from the other CFA. In particular, in our example, instead of first generating trees with left branches only until the partial fixpoint is found, the extended abstraction will immediately take into account also the trees with nodes having some right successors, and hence the final fixpoint will be reached in much fewer steps.

Formally, we first define auxiliary sets $\mathbf{Q}_k$ and $\mathbf{\Delta}_k$ for a fixed SCFA $\mathcal{S}$, a fixed number of components n, a fixed port specification R, and $1 \leq k \leq n$:

$$\begin{aligned} \mathbf{Q}_k &= \{q \in Q'_k : (\mathcal{A}'_1, \ldots, \mathcal{A}'_n, R) \in \mathcal{S}, \mathcal{A}'_i = (\Sigma, Q'_i, \Delta'_i, F'_i)\} \text{ and} \\ \mathbf{\Delta}_k &= \{t \in \Delta'_k : (\mathcal{A}'_1, \ldots, \mathcal{A}'_n, R) \in \mathcal{S}, \mathcal{A}'_i = (\Sigma, Q'_i, \Delta'_i, F'_i)\}. \end{aligned}$$

Intuitively, $\mathbf{Q}_k$ contains union of states of all components with index k which appear within FA having n components and specification R. Similarly $\mathbf{\Delta}_k$ contains union of transitions of all such components. In the next step, for the

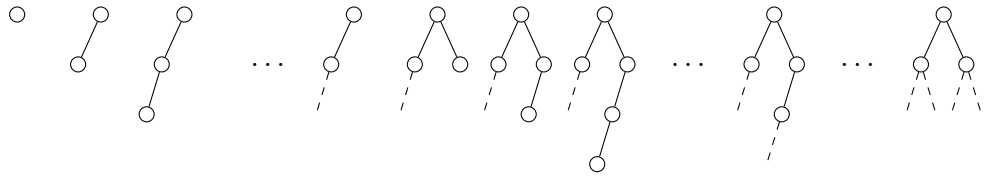


Figure 4.11: A possible evolution of the set of reachable states when the basic abstraction is applied. The dashed lines represent the fact that the structure can be extended by repeating the pattern arbitrary many times.

given CFA $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R) \in \mathcal{S}$ where $\mathcal{A}_i = (\Sigma, Q_i, \Delta_i, F_i)$ for $1 \leq i \leq n$, we compute the *enriched* CFA $\mathcal{F}'$ using the following formula:

$$\mathcal{F}' = ((\Sigma, \mathbf{Q}_1, \mathbf{\Delta}_1, F_1), \ldots, (\Sigma, \mathbf{Q}_n, \mathbf{\Delta}_n, F_n), R).$$

We finish the process by applying the abstraction described in Section 4.4.1 on $\mathcal{F}'$ to obtain $\mathcal{F}'/{\approx}$.

An experimental comparison of the basic and the extended abstraction is provided in Section 4.6.3. The results show that the extended abstraction can indeed significantly improve the speed of the analysis.

4.5 Towards Abstraction Refinement

Whenever our verification procedure reaches a counterexample suggesting that the program being verified contains an error, it should be checked whether the counterexample is not caused by an excessive over-approximation of the reachable state space. In order to do this, one can try to execute the program along the suspected trace without abstraction, and if the error line is still reachable, the program indeed contains an error. If the entire error trace cannot be executed, the counterexample is spurious, in which case we can globally refine the bounded height abstraction by increasing the abstraction height by one and restart the verification process. This kind of refinement, however, does not work very well in practice as it often only leads to obtaining a different (bigger) spurious counterexample. Moreover, the efficiency of the abstraction rapidly falls as the height increases.

A better refinement can be obtained by using the framework of counterexample-guided abstraction refinement together with predicate language abstraction (see [BHV04, BHRV06b] or Section 2.5). For that to work, one needs to be able to execute the program backwards along the suspected trace and see where the forward abstract execution and the backward concrete execution stop having anything in common (meaning that at this point the languages of the automata obtained forward and the languages of the automata obtained backward start having the empty intersection). One can then get back to the last point where the intersection was nonempty, and use languages of the states of the automata appearing at this point as new predicate languages (for more

details, see again [BHV04, BHRV06b])[8]. In what follows, we discuss how a program path suspected to lead to an error may be executed backwards in order to check whether it really leads to an error or whether it corresponds to a spurious counterexample (implying a need to refine the abstraction used).

4.5.1 Symbolic Execution Revisited

First, let us briefly recall some essentials of the (forward) symbolic execution of a program over FA as a basis for backward symbolic execution. The statements of the program are processed sequentially. The symbolic execution of a statement over a set of configurations represented by a CFA can cause the computation to branch. This happens either due to several possible outcomes of the statement when it is executed over the given set of configurations and/or due to the procedure of transforming an FA obtained as a result of the statement into an SCFA (which is then further processed element-wise, hence the branching). The branching creates an execution tree whose nodes are labelled by CFA and the appropriate control line. The nodes of the tree where the computation can still continue are kept in memory together with some additional information about the statements that generated the particular CFA. This additional information, which we will describe in more detail in the following, allows us to reverse the effect of the corresponding statement during the backward symbolic execution. Once the forward symbolic execution hits an erroneous configuration, the program trace suspected to lead to an error can be extracted by traversing the execution tree from the leaf node in which the (possible) error was detected towards the root.

More precisely, assume that we have a suspected error trace—a branch of the execution tree having the form $\mathcal{F}_0, \ldots, \mathcal{F}_n$ where for each $0 \leq i \leq n$, $\mathcal{F}_i$ is a CFA encoding a set of possible configurations of the heap. Here, $\mathcal{F}_0$ represents the set of initial configurations (in our case, the empty heap), and $\mathcal{F}_n$ encodes a set of configurations that include some bad configurations. This means that there is a nonempty set $Bad \subseteq [\![\mathcal{F}_n]\!]$ of hypergraphs that are erroneous meaning that they, e.g., represent heaps with garbage, heaps where a value of a variable to be dereferenced is `null` or `undef`, heap configurations reached at a designated error location, etc. Assume that we have a CFA $\mathcal{F}_{Bad}$ such that $[\![\mathcal{F}_{Bad}]\!] \cap Bad \neq \emptyset$. Such a CFA is obtained either as the result of symbolically executing a statement leading to a designated error line, or, in the case of generic pointer manipulation errors (such as null/undefined dereference, presence ofgarbage), as the result of simply firing some pointer manipulating statement. In the latter case, the way, in which we symbolically execute pointer manipulating statements, guarantees that a CFA encoding all the configurations over which it is unsafe to execute the given statement gets generated. If an SCFA for the bad configurations is obtained, it can be processed element-wise. Further, we know that for

[8] Let us briefly recall that in case of predicate language abstraction, one has a set of predicate languages (represented using automata). A state is said to satisfy some predicate language if the intersection of the language of this state and the the given predicate language is nonempty. Then, the abstraction allows one to collapse the states that satisfy the same predicate languages.

every $1 \leq i \leq n$, the CFA $\mathcal{F}_i$ was obtained from $\mathcal{F}_{i-1}$ by symbolically executing some program statement in the following four steps described in Section 4.1 and also in Section 4.2:

1. materialisation (unfolding boxes represented by SCFA at the relevant FA transitions),
2. introduction of auxiliary roots into the represented heaps (and hence component TA on the level of FA),
3. performing the actual update,
4. iterative folding, normalisation, and abstraction yielding an SCFA.

4.5.2 Backward Symbolic Execution

As usual in counterexample guided abstraction refinement, a backward execution of a program trace computes a chain $\mathcal{F}_{Bad} = \mathcal{F}'_n, \mathcal{F}'_{n-1}, \ldots, \mathcal{F}'_j$ of CFA representing sets of configurations where $j < n$ and either $j \geq 0$ is the greatest index such that $[\![\mathcal{F}_j]\!] \cap [\![\mathcal{F}'_j]\!] = \emptyset$ or $j = 0$ if there is no such index. For each $j < i \leq n$, $\mathcal{F}'_{i-1}$ is derived from $\mathcal{F}'_i$ by applying backwards the steps by which $\mathcal{F}_i$ was obtained from $\mathcal{F}_{i-1}$, but without abstraction. If $j = 0$ and $[\![\mathcal{F}'_0]\!] \cap [\![\mathcal{F}_0]\!] \neq \emptyset$, then the backward execution reached the initial configurations which means that the error trace is feasible in the verified program. If $[\![\mathcal{F}'_j]\!] \cap [\![\mathcal{F}_j]\!] = \emptyset$ for some $j > 0$, then the counterexample is spurious, and $\mathcal{F}'_{j+1}$ represents some configurations introduced by abstraction that caused the discovery of the spurious counterexample. In that case, we may use the pair $\mathcal{F}'_{j+1}, \mathcal{F}_{j+1}$ to derive new predicate languages to refine the abstraction to prevent this spurious error trace from appearing in further verification as described in [BHV04, BHRV06a] and also briefly sketched in Section 2.5.

To be able to revert Steps 1 to 4 of the symbolic execution of a program statement in order to obtain $\mathcal{F}'_i$ from $\mathcal{F}'_{i+1}$, additional information is stored along the computation path. More precisely, we remember where we performed folding and unfolding (to revert Step 1 and Step 4). We also remember information needed to revert the actual update (Step 3) as described later on and the actions which were taken in order to transform the obtained FA into CFA (Step 4). In particular, the following information is recorded for the different supported statements:

- `x = y`, `x = null`, `x = y->s`: We record the previous value of variable `x` such that we can later restore its contents in the configuration obtained during the backward symbolic execution.
- `x->s = y`, `x->s = d`: We again keep the original value of the corresponding field `x->s` such that it can be restored if necessary.
- `malloc(x)`: We remember the previous value of variable `x`. In addition, we record which component was created by `malloc(x)` such that it could be removed when computing backwards.

- `free(x)`: In this case, we need to record the content of the whole node which has been deleted by the statement. The reversal of this statement is then similar to executing `malloc(x)`, but the newly created node already contains initialised fields.

Now, assume that we have computed $\mathcal{F}'_{i+1}$ in the backward execution and we want to revert the effect of the i-th statement. First of all, we have to revert folding and normalisation performed as the last step of the forward execution. The folding is reverted by simply unfolding the corresponding box. The normalisation is reverted by splitting certain component TA (if it was recorded that some component TA were removed during the forward execution) and by an inverse reordering of the component TA (the reordering is also recorded during the forward execution). Next, we revert the effect of the actual update as described above. Step 2 only introduces auxiliary roots which are redundant w.r.t. the canonical representation. Therefore, they can be removed by performing the standard normalisation procedure which converts the given FA into a CFA. Finally, the effect of unfolding performed in Step 1 is reverted by folding if needed.

Checking Intersection of FA. Checking emptiness of the intersection $[\![\mathcal{F}_i]\!] \cap [\![\mathcal{F}'_i]\!]$ is an issue by itself. For two general hierarchical FA $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ and $\mathcal{F}' = (\mathcal{B}_1, \ldots, \mathcal{B}_n, R)$, we do not know yet whether the intersection emptiness problem is decidable. However, we can solve the problem of intersection analogically as in the case of checking language inclusion by using a safe approximation. For two FA $\mathcal{F}$ and $\mathcal{F}'$ having the same root interconnection graph, we can compute the automaton $\mathcal{F} \cap \mathcal{F}' = (\mathcal{A}_1 \cap \mathcal{B}_1, \ldots, \mathcal{A}_2 \cap \mathcal{B}_2, R)$ where $\mathcal{A}_i \cap \mathcal{B}_i$ is the usual intersection of two tree automata. It obviously holds that $[\![\mathcal{F} \cap \mathcal{F}']\!] \subseteq [\![\mathcal{F}]\!] \cap [\![\mathcal{F}']\!]$.

Notice that by having a method that under-approximates the intersection, the only thing that can happen is that emptiness of the intersection $[\![\mathcal{F}_i]\!] \cap [\![\mathcal{F}'_i]\!]$ is detected sooner during the backward execution (for a larger i) than it should be, or it is detected in cases where it should not be detected at all. The only possible consequence of this is that we attempt to refine the abstraction instead of signalling a real counterexample or that we refine the abstraction in a wrong way. Both of the cases can cause the computation not to terminate, but they cannot lead to introducing false positives nor false negatives.

4.6 Experimental Evaluation

We have implemented the proposed approach (up to a backward symbolic execution and automatic abstraction refinement) 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 [ABH+08, ACH+10], a part of that is also described later on in Chapter 5, Chapter 6, and Chapter 7).

Table 4.1: A comparison of Forester (using predefined boxes) with other existing tools

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	
$\text{SLL}_{\text{Linux}}$	< 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 $\text{2CDDLs}_{\text{Linux}}$	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

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 run our tests on a machine with an Intel T9600 (2.8GHz) CPU and 4GiB of RAM.

If not stated otherwise, we used $\approx_1 \cap \approx_C$ for abstraction together with our abstraction scheme for SCFA.

4.6.1 Comparison to Existing Tools

We have experimentally compared the performance of our tool with that of Space Invader [BCC+07], the first fully automated tool based on separation logic, Predator [DPV11], 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 [BHRV06b] based on abstract regular tree model checking[9]. 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 shows up, for example, in the test case with a list of sublists of lengths 0 or 1 for which Space Invader does not terminate. Our technique handles this example smoothly (without any need to add any

[9] Since it is quite difficult to encode the input for ARTMC, we have tried it on some interesting cases only.

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[10]. 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 representation of heap configurations used in ARTMC is much heavier which causes ARTMC not to scale that well.

Table 4.1 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 [DPV11] which 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). Forester has been provided a set of suitable boxes for each of the test cases for the comparison in Table 4.1.

4.6.2 Folding of Nested FA

In addition to the comparison of Forester and some of its competitors, we have also compared the performance of two variants of Forester. In the first case, the user has to provide the boxes which are required for the successful verification of the given program (this variant was used when comparing with other tools and originally published in [HHR+11a]). In the second case, the boxes are learnt fully automatically using the approach presented in Section 4.3. Table 4.2 shows the running times (in seconds) of the two variants as well as the number of boxes which were supplied (learnt).

We can see that the overhead caused by the box discovery algorithm is quite reasonable in most cases. Whenever no nested structures are required for successful verification, the slowdown is usually either not noticeable at all, or it amounts to a small fraction of the original running time. When simple nested structures such as those involved in DLLs are required then the slowdown is

[10] The operations were carefully tuned not to easily generate false alarms, but the risk of generating them has anyway been increased.

Table 4.2: An evaluation of the performance of automatic box discovery

example	predef. boxes		automatic	
	time	boxes	time	boxes
SLL (delete)	0.01	-	0.01	-
SLL (reverse)	< 0.01	-	< 0.01	-
SLL (bubblesort)	0.02	-	0.02	-
SLL (insertsort)	0.02	-	0.02	-
SLL (mergesort)	0.07	-	0.07	-
SLL of CSLLs	0.07	3	0.64	4
SLL+head	0.01	-	0.02	-
SLL of 0/1 SLLs	0.02	-	0.02	-
SLL_{Linux}	< 0.01	-	< 0.01	-
DLL (insert)	0.02	1	0.03	1
DLL (reverse)	0.01	1	0.02	1
DLL (insertsort1)	0.20	1	0.23	1
DLL (insertsort2)	0.06	1	0.08	1
CDLL	< 0.01	1	0.01	1
DLL of CDLLs	0.18	8	0.96	7
SLL of $2CDDLs_{Linux}$	0.03	13	0.09	5
tree	0.06	-	0.10	-
tree+stack	0.02	-	0.04	-
tree+parents	0.10	2	0.16	2
tree (DSW)	0.16	-	0.30	-

also relatively low. The only cases in which we obtained a significant slowdown are some of those which required us to learn hierarchically nested structures. However, according to our analysis, the slowdown in these cases is not caused by some fundamental limitation of the automatic discovery approach, but it is rather due to the heuristics which control when and what nested structures are allowed to be folded. A further improvement of these heuristics is thus an interesting subject of the future work.

Table 4.2 also shows how many boxes (if any) were used for successful verification. We can see that in more complex settings, the automatic approach tends to use different (usually smaller) set of nested structures than what the user typically provides manually. The main reason of this phenomenon is the fact that the user provides the nested structures in the light of what data structures the program is supposed to work with. On the other hand, the discovery heuristics work with the actual configurations appearing during the verification run, and these configurations are not always in a "nice" form as the user would expect them to be.

4.6.3 Abstraction

We have also benchmarked the performance of our tool when different types of abstraction are used. More precisely, we have compared the basic abstraction which processes each CFA within SCFA separately to the more advanced abstraction for SCFA described in Section 4.4.5. The verification times obtained within this comparison are shown in Table 4.3. One can immediately see that

Table 4.3: A comparison of the basic and the SCFA abstraction (time in seconds)

example	abstraction	
	basic	SCFA
SLL (delete)	0.01	0.01
DLL (reverse)	0.01	0.02
SLL of CSLLs	1.75	0.64
DLL of CDLLs	4.31	0.96
tree (DSW)	1.11	0.30

the SCFA abstraction indeed substantially helps in more complicated cases. On the other hand, for simple data structures—such as plain singly/doubly-linked lists—the verification times remain unchanged, or the SCFA abstraction is a bit slower as its application incurs some overhead. The overhead is, however, so small that we by default use the SCFA abstraction in all further experiments.

In addition, we have also tested the behaviour of our tool when $\approx_1 \cap \approx_{C\uparrow}$ is used for abstraction (instead of $\approx_1 \cap \approx_C$) even in benchmarks for which it is not needed, however, the running times for the considered cases were identical.

4.6.4 Additional Experiments

On top of the programs considered in the paper [HHR$^+$11a] on which this chapter is mostly based, we have later tried some additional challenging programs. In particular, we will now briefly discuss our experiments with *skip lists* to further highlight capabilities of our box discovery procedure. On this example, we also illustrate an improvement in scalability which can be achieved by using language inclusion instead of language equality when one matches a newly discovered box against the set of already existing boxes. We will also discuss our initial attempts on the verification of recursive programs.

Verification of Skip Lists. A skip list (see [Pug90]) is usually a randomised data structure which allows efficient insertion and retrieval of data such that it provides an alternative to various balanced trees. A skip list can be seen as a multi-layered singly-linked list such that each element of the list contains multiple next pointers. Each layer of the list forms a traditional singly-linked list such that elements present in the list of level $i + 1$ are also present in the list of level i. The list of level 0 always contains all elements of the list. To find a key k in such a data structure, one proceeds as follows. First, one starts by traversing the top-level list until one finds an element whose successor contains a key which is bigger than k. Then, one descends by one level and continues the traversal until one finds an element which is bigger than k. Then, one descends again and repeats the whole process until level 0 is reached. At that point, one has either found an element corresponding to k or such element is not present within the skip list. During insertion of a new element, the skip list is traversed in the same manner as in the case of a look-up. When the algorithm finds a position at level 0 where the new element should be inserted,

Table 4.4: A comparison of the two box matching approaches

example	equivalence		inclusion	
	time	boxes	time	boxes
skip list (2 levels)	0.65 s	3	0.30 s	2
skip list (3 levels)	17.65 h	278	10.00 s	5
SLL of CSLLs	0.64 s	4	0.55 s	3
DLL of CDLLs	0.96 s	7	0.92 s	7
SLL of $2CDDLs_{Linux}$	0.09 s	5	0.09 s	5

it non-deterministically chooses up to what level i the element should be linked and then links it into all levels $0, \ldots, i$.

Formal verification of programs manipulating general skip lists is quite difficult. In our framework, a set of skip lists, in which the number of layers is not limited, contains an unbounded number of cut-points which cannot be eliminated by our cut-point elimination procedure. To the best of our knowledge, however, no sufficiently automatic method for unbounded skip lists has been developed so far. One can mention [MTLT10] and [CRN07], in which the authors consider a verification of skip lists with one layer of skipping (i.e., with 2 levels altogether), and they both require that the user provides additional information alongside the verified code.

In order to verify a program working with skip lists (in particular, a program performing repeated random insertion), we have also done certain modifications to the original algorithm. First, we have fixed the number of levels to 2 and 3 (in 2 different experiments), hence consisting of 1 or 2 layers of skipping, repectively. Moreover, we have slightly modified the original code such that the memory safety is no longer requiring the data stored in the skip lists be sorted which we currently do not handle[11].

Table 4.4 shows the running times and the number of boxes which were required in order to successfully verify the considered program performing repeated insertion of elements into skip lists with 2 or 3 levels (as we have already mentioned, 3 level skip lists required $\approx_1 \cap \approx_{C\uparrow}$). In addition, we also provide some selected benchmarks from Table 4.2. The column labelled by "equivalence" contains data for verification runs in which we only allowed a box to be folded when the set of substructures appearing within the given configuration matches exactly the content of the box in the database. On the other hand, the column labelled by "inclusion" lists data for the verification runs in which we allowed a box to be folded even if only a subset of the substructures represented by the box appears within the given configuration.

First, we can see that we can verify 2 level skip lists quite efficiently no matter of what kind of matching we use. However, this is not the case when one proceeds to skip lists with 3 levels. The variant with exact matching is substan-

[11] The skip list insertion/lookup algorithms rely on that the elements in the lowest level are ordered. Due to that, a search/insertion on level $i-1$ can never leave from within the selected segment of level i (delimited by two elements linked at level i one of which is smaller and the other one bigger than the element being sought/inserted). As a workaround, we have modified the algorithm to explicitly check that the search on level $i-1$ never leaves the selected segment at level i.

```
struct Item* find(struct Item* head, int key) {
   if (head == NULL)
     return NULL;
   if (head->key == key)
     return head;
   return find(head->next, key);
}
```

Figure 4.12: A simple recursive C function for finding an element within a singly-linked list

tially slower. In this case, Forester needs to first fold segments of (sub-)skip lists appearing at level 0 and 1 into boxes. These boxes are later hierarchically folded into boxes representing segments of skip lists with all 3 layers. As we can see from Table 4.4, 3 boxes are needed already for 2 levels which causes that the number of boxes for representing 3 levels blows up. Fortunately, when one switches to box inclusion, the situation improves substantially. The first reason is that we only need 2 boxes for verification of 2 level skip lists, therefore the blowup in the number of boxes is reduced. Second, by using inclusion instead of equivalence, we in fact allow another form of abstraction, which helps Forester to converge faster. More precisely, imagine that Forester performs an abstraction of some state at some moment. Then, it can happen that a part of the abstracted automaton is used to form a box. When this box is later folded (instead of a smaller one), the folded part of the automaton gets automatically abstracted even in the case in which the ordinary abstraction would not collapse any states. Finally, by keeping only the boxes which are maximal w.r.t. the language inclusion, we further reduce the total number of boxes required for successful verification.

Verification of Recursive Programs. As we have already mentioned in Section 4.1.2, we encode the call stack using a special singly-linked list stored directly inside the heap. This opens a possibility to verify simple recursive programs. Consider, for instance, the recursive function `find` in Figure 4.12 which searches for a given element inside a singly-linked list.

We can immediately see that each call of `find` creates two variables stored on the stack (here declared as formal parameters) out of which the first one points to an element of the list. After a brief analysis, one can observe that in our encoding, `find` in fact creates a certain kind of grid as depicted in Figure 4.13.

Unfortunately, the corresponding class of graphs is beyond what we can currently handle. However, we can observe that the complexity of the data structure is caused by the precise representation of the content of the stack. Indeed, the content of the variable `head` is actually never used after the recursive call to `find` and can be discarded. This fact, which can be detected by a simple static live variable analysis, can be used to allow Forester to handle recursive functions such as `find`. In particular, in case of `find`, the content of variable

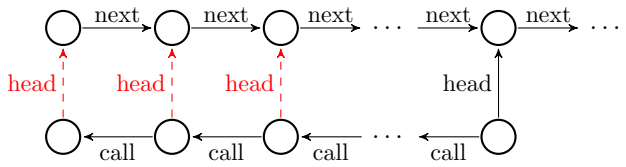


Figure 4.13: Heap configurations arising during the verification of `find`. The first row of nodes represents a singly-linked list which is to be traversed. The content of the call stack is represented by the second row in which each node corresponds to a single call. The picture also demonstrates that each instantiation of the local variable `head` points to a different element of the list.

`head` gets erased immediately after the value is passed into the recursive call (see the dashed links highlighted in red in Figure 4.13).

Forester implements a traditional live variable analysis (see [NNH99]) which is used to remove the content of dead variables as soon as possible. When using this extension, the verification of `find` takes about 0.02 s.

In addition, removing the content of dead variables typically allows Forester to reduce the number of generated FA (differing just in the values of dead variables). Hence, the efficiency of Forester is improved even in cases in which there is no recursion at all.

4.7 Related Work

The area of verifying programs with dynamically linked data structures has been a subject of intense research for quite some time. Many different approaches based on the various kinds of logics, e.g., [MS01, SRW02, Rey02, BCC+07, GVA07, NDQC07, CRN07, ZKR08, YLB+08, CDOY09, MTLT10, MPQ11, DPV11], automata [BHRV06b, BBH+11, DEG06], upward closed sets [ABC+08, ACV11], and other formalisms have been proposed. These approaches differ in their generality, efficiency, and degree of automation. We concentrate on a comparison with the two closest lines of work, namely, the use of automata as described in [BHRV06b] and the use of separation logic in the works [BCC+07, YLB+08] linked with Space Invader and later Slayer and to some extent Predator. In fact, as we have already noted earlier, the approach we propose combines some features from these two lines of research.

Compared to [BCC+07, YLB+08], 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., [NDQC07], that consider tree manipulation, but these are usually semi-automated only. An exception is [GVA07] 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[12]). An approach presented in [MTLT10] can also handle tree

[12] We did not find an available implementation of [GVA07], and so we could not try it out ourselves.

structures, but their method requires substantial user aid in the form of recursive predicates in separation logic which are linked to the structure of the code to be verified (in fact this is very close to providing inductive invariants of the program). Therefore, unlike in our case, the method is not fully automatic.

Further, compared to [BCC+07, YLB+08], 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 [YLC+07]. In such cases, the abstraction used in [BCC+07, YLB+08] 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 [BHRV06b], 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 [BHRV06b]. This is due to the fact that the heap representation in [BHRV06b] 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 [BHRV06b] 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 [BHRV06b] has troubles with detection of memory leakage, which is in theory possible, but it is so complex that it has never been implemented.

4.8 Conclusions and Future Work

In this chapter, we have proposed a new method for verification of heap manipulating programs. In particular, we use forest automata proposed in Chapter 3 to encode sets of heaps and exploit the fact that the set of C statements that we need to support can be easily symbolically executed over FA. Moreover, the fact that FA are built over tree automata allows us to build a verification procedure based on the framework of abstract regular tree model checking. For use in our setting, we have proposed specialisations of the general-purpose abstraction schemas, which have been introduced in ARTMC. Further, we have described how more complex data structures—such as doubly-linked lists—can be veri-

fied using hierarchically nested forest automata whose hierarchical structuring can be discovered fully automatically. Finally, we have implemented the above mentioned approach in a prototype tool called Forester. We have performed an experimental evaluation on a set of benchmarks consisting of C programs manipulating various dynamically allocated data structures in order to compare Forester to other similar tools. The obtained results confirmed that our approach is quite competitive in practice.

As of what concerns the future work, one of the most interesting areas is an implementation of the proposed but not yet implemented abstraction refinement which relies on the ability to perform a backward execution along the trace that seems to lead to an error state. A part of this work should be an evaluation of whether the proposed under-approximation of intersection is sufficient in practice, or whether some more precise approach is needed.

Furthermore, it would also be interesting to extend our approach such that it could track some information about the data stored within dynamically linked data structures. This would allow one to verify algorithms in which the memory safety depends, for instance, on the fact that a certain sequence is sorted. As an example, we can mention the algorithm for skip lists which we had to manually modify in order to remove the dependency on the data. Another example is that of dealing with red-black trees in which case one needs to distinguish red and black trees. Apart from that, tracking of the data stored inside the dynamically linked data structure would allow Forester to also check properties concerning that data.

Another line of research is a generalisation of our approach to concurrent programs. Here, an especially interesting case is that of lockless concurrent data structures, which are extremely difficult to understand and validate.

Finally, some programs manipulating complex data structures—such as trees with linked leaves—could be verified if one was able to work with recursively nested boxes. Therefore, a generalisation of the algorithms proposed in this thesis for such boxes is also an interesting topic for future work.

5 Simulations over LTSs and Tree Automata

The approach of abstract regular (tree) model checking, which we use in a novel way also in our verification technique for programs with dynamic linked data structures proposed in the previous chapter, crucially depends on the efficiency of dealing with automata. AR(T)MC ([BHV04, BHRV06a]) was originally built over deterministic finite tree automata. The need to determinise the automata in every step of the computation has, however, turned out to be a significant obstacle to practical applicability of the approach. That is why, in [BHH+08], it has been proposed to replace their use by nodeterministic tree automata (NTA). This, however, brings some problems to be solved. In particular, one needs to be able to perform inclusion checking (in order to see when a fixpoint is reached) and to reduce the of the automata obtained in the computation. Unfortunately, both standard minimisation and inclusion checking algorithms are based on first making the appropriate automata deterministic. Using determinisation as an intermediate step would, however, destroy the advantage of working with usually much smaller NTA. Hence, both of the operations are to be done without determinisation. For checking inclusion, one can use methods based on antichains, possibly combined with simulation as discussed in Chapter 6 and in Chapter 7. For reducing the size of the automata, one can use quotienting w.r.t. a suitable, typically simulation-based equivalence relation (see [ABH+08]). For both of these problems, it is thus crucial to be able to efficiently compute simulations on NTA. One of the most efficient ways to obtain these equivalence simulation relations on NTA is via translating an NTA into a labelled transition system (LTS) as described in [AHKV08] and then computing the simulation relation on this LTS.

Apart form that, many other automated verification techniques—such as LTL model checking—are also directly or indirectly dealing with LTSs and as such they are often limited by their size. One of the well-established approaches to cope with this problem is the reduction of an LTS using a suitable equivalence relation according to which the states of the LTS are collapsed. A good candidate for such a relation is again simulation equivalence. It strongly preserves logics like $ACTL^*$, $ECTL^*$, and LTL [DGG93, GL94, HHK95], and with respect to its reduction power and computation cost, it offers a desirable compromise among the other common candidates, such as bisimulation equivalence [PT87, SJ05] and language equivalence.

The currently fastest LTS-simulation algorithm (below denoted as LRT—i.e., labelled RT) has been published in [ABH+08]. It is a straightforward modification of the fastest algorithm (in the following denoted as RT, standing for Ranzato-Tapparo) for computing simulations over Kripke structures [RT07],

which itself improves the algorithm from [HHK95]. The time complexity of RT amounts to $\mathcal{O}(|P_{Sim}||\delta|)$, the space complexity amounts to $\mathcal{O}(|P_{Sim}||S|)$. In the case of LRT, the time complexity is $\mathcal{O}(|P_{Sim}||\delta| + |\Sigma||P_{Sim}||S|)$ and the space complexity is $\mathcal{O}(|\Sigma||P_{Sim}||S|)$. Here, S is the set of states of an LTS, δ is its transition relation, Σ is its alphabet, and P_{Sim} is the partition of S according to the simulation equivalence. The space complexity blow-up of LRT is caused by indexing the data structures of RT by the symbols of the alphabet.

In this chapter, we propose an optimised version of LRT (denoted OLRT) that lowers the above described blow-up. We exploit the fact that not all states of an LTS have incoming and outgoing transitions labelled by all symbols of the alphabet, which allows us to reduce the memory footprint of the data structures used during the computation. Our experiments show that the optimisations we propose lead to significant savings of space as well as of time in many practical cases. Moreover, we have achieved a promising reduction of the asymptotic complexity of algorithms for computing tree-automata simulations from [ABH+08] using OLRT, too.

Plan of the Chapter. The text is organised as follows. In Section 5.1, we introduce additional preliminaries used throughout the rest of this chapter. Section 5.2 presents the original algorithm upon which our optimisations are built, In Section 5.3, we describe the proposed optimisations and we derive the complexity of our optimised algorithm. Section 5.4 discusses the complexity of computing simulations over tree automata when our optimised algorithm is used as a part of the procedure. The experimental evaluation of our optimisations is provided in Section 5.5. Finally, Section 5.6 concludes the chapter.

5.1 Preliminaries

Given a binary relation ρ over a set X, we use $\rho(x)$ to denote the set $\{y \mid (x, y) \in \rho\}$. Then, for a set $Y \subseteq X$, $\rho(Y) = \bigcup\{\rho(y) \mid y \in Y\}$. A *partition-relation pair* over X is a pair $\langle P, Rel\rangle$ where $P \subseteq 2^X$ is a partition of X (we call elements of P *blocks*) and $Rel \subseteq P \times P$. A partition-relation pair $\langle P, Rel\rangle$ *induces* the relation $\rho = \bigcup_{(B,C)\in Rel} B \times C$. We say that $\langle P, Rel\rangle$ is the *coarsest partition-relation pair* inducing ρ if any two $x, y \in X$ are in the same block of P if and only if $\rho(x) = \rho(y)$ and $\rho^{-1}(x) = \rho^{-1}(y)$. Note that in the case when ρ is a preorder and $\langle P, Rel\rangle$ is coarsest, then P is the set of equivalence classes of $\rho \cap \rho^{-1}$ and Rel is a partial order.

For an LTS $T = (S, \Sigma, \{\delta_a \mid a \in \Sigma\})$, a *simulation* over T is a binary relation ρ on S such that if $(u, v) \in \rho$, then for all $a \in \Sigma$ and $u' \in \delta_a(u)$, there exists $v' \in \delta_a(v)$ such that $(u', v') \in \rho$. It can be shown that for a given LTS T and an *initial preorder* $I \subseteq S \times S$, there is a unique maximal simulation Sim_I on T that is a subset of I, and that Sim_I is a preorder (see [ABH+08]).

5.2 The Original LRT Algorithm

In this section, we describe the original version of the algorithm presented in [ABH+08], i.e., the algorithm that we denote as LRT (see Algorithm 3).

The algorithm gradually refines a partition-relation pair $\langle P, Rel\rangle$, which is initialised as the coarsest partition-relation pair inducing an initial preorder I. After its termination, $\langle P, Rel\rangle$ is the coarsest partition-relation pair inducing Sim_I. The basic invariant of the algorithm is that the relation induced by $\langle P, Rel\rangle$ is always a superset of Sim_I.

The while-loop refines the partition P and then prunes the relation Rel in each iteration of the while-loop. The role of the *Remove* sets can be explained as follows: During the initialisation, every $Remove_a(B)$ is filled by states v such that $\delta_a(v) \cap \bigcup Rel(B) = \emptyset$ (there is no a-transition leading from v "above" B w.r.t. Rel). During the computation phase, v is added into $Remove_a(B)$ after $\delta_a(v) \cap \bigcup Rel(B)$ becomes empty (because of pruning Rel on line 19). Emptiness of $\delta_a(v) \cap Rel(B)$ is tested on line 22 using counters $Count_a(v, B)$, which record the cardinality of $\delta_a(v) \cap Rel(B)$. From the definition of simulation, and because the relation induced by $\langle P, Rel\rangle$ is always a superset of Sim_I, $\delta_a(v) \cap \bigcup Rel(B) = \emptyset$ implies that for all $u \in \delta_a^{-1}(B)$, $(u, v) \notin Sim_I$ (v cannot simulate any $u \in \delta_a^{-1}(B)$). To reflect this, the relation Rel is pruned each time $Remove_a(B)$ is processed. The code on lines 9–15 prepares the partition-relation pair and all the data structures. First, $Split(P, Remove_a(B))$ divides every block B' into $B' \cap Remove_a(B)$ (which cannot simulate states from $\delta_a^{-1}(B)$ as they have empty intersection with $\delta_a^{-1}(Rel(B))$), and $B' \setminus Remove_a(B)$. More specifically, for a set $Remove \subseteq S$, $Split(P, Remove)$ returns a finer partition $P' = \{B \setminus Remove \mid B \in P\} \cup \{B \cap Remove \mid B \in P\}$. After refining P by the *Split* operation, the newly created blocks of P inherit the data structures (counters *Count* and *Remove* sets) from their "parents" (for a block $B \in P$, its parent is the block $B_{\mathsf{prev}} \in P_{\mathsf{prev}}$ such that $B \subseteq B_{\mathsf{prev}}$). Rel is then updated on line 19 by removing the pairs (C, D) such that $C \cap \delta_a^{-1}(B) \neq \emptyset$ and $D \subseteq Remove_a(B)$. The change of Rel causes that for some states $u \in S$ and symbols $b \in \Sigma$, $\delta_a(u) \cap \bigcup Rel(C)$ becomes empty. To propagate the change of the relation along the transition relation, u will be moved into $Remove_b(C)$ on line 23, which will cause new changes of the relation in the following iterations of the while-loop. If there is no nonempty *Remove* set, then $\langle P, Rel\rangle$ is the coarsest partition-relation pair inducing Sim_I and the algorithm terminates. Correctness of LRT is stated by Theorem 1.

Theorem 1 ([ABH+08]) *With an LTS $T = (S, \Sigma, \{\delta_a \mid a \in \Sigma\})$ and the coarsest partition-relation pair $\langle P_I, Rel_I\rangle$ inducing a preorder $I \subseteq S \times S$ on the input, LRT terminates with the coarsest partition-relation pair $\langle P, Rel\rangle$ inducing Sim_I.*

5.3 Optimisations of LRT

The optimisation that we are now going to propose reduces the number of counters and the number and the size of *Remove* sets. The changes required

Algorithm 3: (O)LRT Algorithm

Input: an LTS $T = (S, \Sigma, \{\delta_a \mid a \in \Sigma\})$, partition-relation pair $\langle P_I, Rel_I \rangle$
Output: partition-relation pair $\langle P, Rel \rangle$

```
   /* initialization */
1  ⟨P, Rel⟩ ← ⟨P_I, Rel_I⟩                          /* ← ⟨P_{I∩Out}, Rel_{I∩Out}⟩ */
2  foreach B ∈ P and a ∈ Σ do                                  /* a ∈ in(B) */
3     foreach v ∈ S do                                         /* v ∈ δ_a^{-1}(S) */
4        Count_a(v, B) = |δ_a(v) ∩ ⋃ Rel(B)|;
5     Remove_a(B) ← S \ δ_a^{-1}(⋃ Rel(B))        /* ← δ_a^{-1}(S)\δ_a^{-1}(⋃ Rel(B)) */
   /* computation */
6  while exists B ∈ P and a ∈ Σ such that Remove_a(B) ≠ ∅ do
7     Remove ← Remove_a(B);
8     Remove_a(B) ← ∅;
9     ⟨P_prev, Rel_prev⟩ ← ⟨P, Rel⟩;
10    P ← Split(P, Remove);
11    Rel ← {(C, D) ∈ P × P | (C_prev, D_prev) ∈ Rel_prev};
12    foreach C ∈ P and b ∈ Σ do                                /* b ∈ in(C) */
13       Remove_b(C) ← Remove_b(C_prev);
14       foreach v ∈ S do                                      /* v ∈ δ_b^{-1}(S) */
15          Count_b(v, C) ← Count_b(v, C_prev);
16    foreach C ∈ P such that C ∩ δ_a^{-1}(B) ≠ ∅ do
17       foreach D ∈ P such that D ⊆ Remove do
18          if (C, D) ∈ Rel then
19             Rel ← Rel \ {(C, D)};
20             foreach b ∈ Σ and v ∈ δ_b^{-1}(D) do      /* b ∈ in(D) ∩ in(C) */
21                Count_b(v, C) ← Count_b(v, C) − 1;
22                if Count_b(v, C) = 0 then
23                   Remove_b(C) ← Remove_b(C) ∪ {v};
```

by our optimised algorithm (OLRT) are indicated in Algorithm 3 on the right hand sides of the concerned lines.

We will need the following notation. For a state $v \in S$, $\mathrm{in}(v) = \{a \in \Sigma \mid \delta_a^{-1}(v) \neq \emptyset\}$ is the set of *input symbols* and $\mathrm{out}(v) = \{a \in \Sigma \mid \delta_a(v) \neq \emptyset\}$ is the set of *output symbols* of v. The *output preorder* is the relation $Out = \bigcap_{a \in \Sigma} \delta_a^{-1}(S) \times \delta_a^{-1}(S)$ (this is, $(u, v) \in Out$ if and only if $\mathrm{out}(u) \subseteq \mathrm{out}(v)$).

To make our optimisation possible, we have to initialise $\langle P, Rel \rangle$ by the finer partition-relation pair $\langle P_{I \cap Out}, Rel_{I \cap Out} \rangle$ (instead of $\langle P_I, Rel_I \rangle$), which is the coarsest partition-relation pair inducing the relation $I \cap Out$. As both I and Out are preorders, $I \cap Out$ is a preorder too. As $Sim_I \subseteq I$ and $Sim_I \subseteq Out$ (any simulation on T is a subset of Out), Sim_I equals the maximal simulation included in $I \cap Out$. Thus, this step itself does not influence the output of the algorithm.

Assuming that $\langle P, Rel \rangle$ is initialised to $\langle P_{I \cap Out}, Rel_{I \cap Out} \rangle$, we can observe that for any $B \in P$ and $a \in \Sigma$ chosen on line 6, the following two claims hold:

Claim 1 *If $a \notin \mathrm{in}(B)$, then skipping this iteration of the while-loop does not affect the output of the algorithm.*

Proof. In an iteration of the while-loop processing $Remove_a(B)$ with $a \notin \text{in}(B)$, as there is no $C \in P$ with $\delta_a(C) \cap Rel(B) \neq \emptyset$, the for-loop on line 18 stops immediately. No pair (C, D) will be removed from *Rel* on line 19, no counter will be decremented, and no state will be added into a *Remove* set. The only thing that can happen is that $Split(P, Remove)$ refines P. However, in this case, this refinement of P would be done anyway in other iterations of the while-loop when processing sets $Remove_b(C)$ with $b \in \text{in}(C)$. To see this, note that correctness of the algorithm does not depend on the order in which nonempty *Remove* sets are processed. Therefore, we can postpone processing all the nonempty $Remove_a(B)$ sets with $a \notin \text{in}(B)$ to the end of the computation. Recall that processing no of these *Remove* sets can cause that an empty *Remove* set becomes nonempty. Thus, the algorithm terminates after processing the last of the postponed $Remove_a(B)$ sets. If processing some of these $Remove_a(B)$ with $a \notin \text{in}(B)$ refines P, P will contain blocks C, D such that both (C, D) and (D, C) are in *Rel* (recall that when processing $Remove_a(B)$, no pair of blocks can be removed from *Rel* on line 19). This means that the final $\langle P, Rel \rangle$ will not be coarsest, which is a contradiction with Theorem 1. Thus, processing the postponed $Remove_a(B)$ sets can influence nor *Rel* neither P, and therefore they do not have to be processed at all. □

Claim 2 *It does not matter whether we assign* $Remove_a(B)$ *or* $Remove_a(B) \setminus (S \setminus \delta_a^{-1}(S))$ *to Remove on line 6.*

Proof. Observe that v with $a \notin \text{out}(v)$ (i.e., $v \in S \setminus \delta_a^{-1}(S)$) cannot be added into $Remove_a(B)$ on line 23, as this would mean that v has an a-transition leading to D. Therefore, v can get into $Remove_a(B)$ only during initialisation on line 5 together with all states from $S \setminus \delta_a^{-1}(S)$. After $Remove_a(B)$ is processed (and emptied) for the first time, no state from $S \setminus \delta_a^{-1}(S)$ can appear there again. Thus, $Remove_a(B)$ contains states from $S \setminus \delta_a^{-1}(S)$ only when it is processed for the first time and then it contains all of them. It can be shown that for any partition Q of a set X and any $Y \subseteq X$, if $Split(Q, Y) = Q$, then also for any $Z \subseteq X$ with $Y \subseteq Z$, $Split(Q, Z) = Split(Q, Z \setminus Y)$. As P refines $P_{I \cap Out}$, $Split(P, S \setminus \delta_a^{-1}(S)) = P$. Therefore, as $S \setminus \delta_a^{-1}(S) \subseteq Remove_a(B)$, $Split(P, Remove_a(B)) = Split(P, Remove_a(B) \setminus (S \setminus \delta_a^{-1}(S)))$. We have shown that removing $S \setminus \delta_a^{-1}(S)$ from *Remove* does not influence the result of the *Split* operation in this iteration of the while-loop (note that this implies that all blocks from the new partition are included in or have empty intersection with $S \setminus \delta_a^{-1}(S)$). It remains to show that the change also does not influence updating *Rel* on line 19. Removing $S \setminus \delta_a^{-1}(S)$ from *Remove* could only cause that the blocks D such that $D \subseteq S \setminus \delta_a^{-1}(S)$ that were chosen on line 17 with the original value of *Remove* will not be chosen with the restricted *Remove*. Thus, some of the pairs (C, D) removed from *Rel* with the original version of *Remove* could stay in *Rel* with the restricted version of *Remove*. However, such a pair (C, D) cannot exist because with the original value of *Remove*, if (C, D) is removed from *Rel*, then $a \in \text{out}(C)$ (as $\delta(C) \cap B \neq \emptyset$) and therefore also $a \in \text{out}(D)$ (as *Rel* was initialised to $Rel_{I \cap Out}$ on line 1 and $(C, D) \in Rel$). Thus, $D \cap (S \setminus \delta_a^{-1}(S)) = \emptyset$, which means that (C, D) is removed from *Rel* even

with the restricted *Remove*. Therefore, it is not important whether $S \setminus \delta_a^{-1}(S)$ is a subset of or it has an empty intersection with *Remove*. □

As justified above, we can optimise LRT as follows. Sets $Remove_a(B)$ are computed only if $a \in \text{in}(B)$ and in that case we only add states $q \in \delta_a^{-1}(S)$ to $Remove_a(B)$. As a result, we can reduce the number of required counters by maintaining $Count_a(v, B)$ if and only if $a \in \text{in}(B)$ and $a \in \text{out}(v)$.

5.3.1 Data Structures

We now describe the essential data structures which are required by OLRT. The input LTS is represented as a list of records about its states. The record about each state $v \in S$ contains a list of nonempty $\delta_a^{-1}(v)$ sets[1], each of them encoded as a list of its members. The partition P is encoded as a doubly-linked list (DLL) of blocks. Each block is represented as a DLL of (pointers to) states of the block. Each block B contains for each $a \in \Sigma$ a list of (pointers on) states from $Remove_a(B)$. Each time when any set $Remove_a(B)$ becomes nonempty, block B is moved to the beginning of the list of blocks. Choosing the block B on line 6 then means just scanning the head of the list of blocks.

Each block $B \in P$ and each state $v \in S$ contains an Σ-indexed array containing a record $B.a$ and $v.a$, respectively. The record $B.a$ stores the information whether $a \in \text{in}(B)$ (we need the test on $a \in \text{in}(B)$ to take a constant time), If $a \in \text{in}(B)$, then $B.a$ also contains a reference to the set $Remove_a(B)$, represented as a list of states (with a constant time addition), and a reference to an array of counters $B.a.Count$ containing the counter $Count_a(v, B)$ for each $v \in \delta_a^{-1}(S)$. Note that for two different symbols $a, b \in \Sigma$ and some $v \in S$, the counter $Count_a(v, B)$ has different index in the array $B.a.Count$ than the counter $Count_b(v, B)$ in $B.b.Count$ (as the sets $\delta_a^{-1}(S)$ and $\delta_b^{-1}(S)$ are different). Therefore, for each $v \in S$ and $a \in \Sigma$, $v.a$ contains an index v_a under which for each $B \in P$, the counter $Count_a(v, B)$ can be found in the array $B.a.Count$. Using the Σ-indexed arrays attached to symbols and blocks, every counter can be found/updated in a constant time. For every $v \in S, a \in \Sigma$, $v.a$ also stores a pointer to the list containing $\delta_a^{-1}(v)$ or *null* if $\delta_a^{-1}(v)$ is empty. This allows the constant time testing whether $a \in \text{in}(v)$ and the constant time searching for the $\delta_a^{-1}(v)$ list.

5.3.2 Complexity of Optimised LRT

Here, we provide a complexity analysis of our optimised algorithm. In order to improve the readability, we fix $IO = I \cap Out$ for the rest of this section. The $Split(P, X)$ operation can be implemented as follows: Iterate through all $v \in X$. If $v \in B \in P$, add v into a block B_{child} (if B_{child} does not exist yet, create it and add it into P) and remove v from B. If B becomes empty, discard it. This can be done in $\mathcal{O}(|X|)$ time.

[1] We use a list rather than an array having an entry for each $a \in \Sigma$ in order to avoid a need to iterate over alphabet symbols for which there is no transition.

Complexity of Initialization Phase (lines 1–5). Computation of $\langle P_{IO}, Rel_{IO}\rangle$ on line 1 can be done in time at most $|\Sigma||P_{IO}|^2$ (starting with $\langle P_I, Rel_I\rangle$, and for each $a \in \Sigma$, splitting P according to $\delta_a^{-1}(S)$ and removing the relation between blocks containing states from $\delta_a^{-1}(S)$ and those containing states from $S \setminus \delta_a^{-1}(S)$). The initialisation of the Σ-indexed arrays attached to states and blocks can be done in $\mathcal{O}(|\Sigma||S| + |\delta|)$ time.

The *Count* counters are initialised by (1) allocating above described arrays of counters (attached to blocks), setting all the counters to 0, and then (2) for all $B \in P$, for all $u \in IO(B)$, and for all $a \in \mathrm{in}(u)$, and for all $v \in \delta_a^{-1}(u)$, incrementing $Count_a(v, B)$. This takes

$$\mathcal{O}\left(\sum_{B\in P_{IO}}\sum_{a\in\mathrm{in}(B)}|\delta_a^{-1}(S)| + \sum_{B\in P_{IO}}\sum_{v\in IO(B)}|\delta^{-1}(v)|\right)$$

time. The *Remove* sets are initialised by iterating through all the counters and if $Count_a(v, B) = 0$, then adding (appending) v to $Remove_a(B)$. This takes time proportional to the number of counters, which is $\mathcal{O}(\sum_{B\in P_{IO}}\sum_{a\in\mathrm{in}(B)}|\delta_a^{-1}(S)|)$. Thus, the overall time complexity of the initialisation is

$$\mathcal{O}\left(|\Sigma||P_{IO}|^2 + |\Sigma||S| + |\sum_{B\in P_{IO}}\sum_{a\in\mathrm{in}(B)}|\delta_a^{-1}(S)| + \sum_{B\in P_{IO}}\sum_{v\in IO(B)}|\delta^{-1}(v)|\right).$$

Complexity of Splitting (lines 10–15). The complexity analysis of lines 12–15 is based on the fact that it can happen at most $|P_{IO}| - |P_{Sim_I}|$ times that a block B is split on line 10. Moreover, the presented code can be optimised by not having the lines 12–15 as a separate loop (this was chosen just for clarity of the presentation), but the inheritance of *Rel*, *Remove*, and the counters can be done within the *Split* function, and only for those blocks that were really split (not for all the blocks every time). Whenever a block B is split into B_1 and B_2, we have to do the following: (1) allocate the Σ-indexed arrays containing the record $B_1.a, B_2.a$ for each $a \in \Sigma$, (the arrays of B can be reused for one of the new blocks); (2) for each $a \in \mathrm{in}(B)$, compute the sets $\mathrm{in}(B)_1$ and $\mathrm{in}(B)_2$ and update the records $B_1.a$ and $B_2.a$ (saying whether $a \in \mathrm{in}(B_1, B,_2)$). This takes time $\mathcal{O}(\sum_{v\in B}|\delta^{-1}(v)|)$ for one block B, which gives time

$$\mathcal{O}\left(\sum_{B\in P_{Sim_I}}\sum_{v\in(IO\cap IO^{-1})(B)}|\delta^{-1}(v)| + |\Sigma|\right)$$

overall; (3) for each $B_i, i \in \{1, 2\}$ and each $a \in \mathrm{in}(B_i)$, copy the $Remove_a(B)$ and the array of the counters $B.a.Count$ and save them to the $B_i.a$ record. The overall time needed for this copying is equal to the overall space taken by all *Remove* sets and all counters, which is $\mathcal{O}(\sum_{B\in P_{Sim_I}}\sum_{a\in\mathrm{in}(B)}|\delta_a^{-1}(S)|)$; (4) add a row and a column to the *Rel* matrix and copy the entries from those of the parent B. This operation takes $\mathcal{O}(|P_{Sim_I}|)$ time for one added block as the size of the rows and columns of the *Rel*-matrix is bounded by $|P_{Sim_I}|$. Thus, for all newly generated blocks, we achieve the overall time complexity of $\mathcal{O}(|P_{Sim_I}|^2)$.

The key observation, which the time complexity analysis of line 10 and lines 16–18 is based on, is the following: For any two states $u, v \in S$ and any symbol $a \in \Sigma$, it can happen at most once that v is present in $\mathit{Remove}_a(B)$ for some block B with $u \in B$ when $\mathit{Remove}_a(B)$ is processed in the main while-loop (B and a is chosen on line 5). This also means that for every $B \in P_{\mathit{Sim}_I}$ and $a \in \Sigma$, the sum of cardinalities of all $\mathit{Remove}_a(B')$ sets with $B \subseteq B'$ (in the moment when a and B' were chosen on line 5) is below $|\delta_a^{-1}(S)|$. Indeed, notice that when v is being added into $\mathit{Remove}_a(B)$ (either on line 5 or on line 23), then $\delta_a(v) \cap \mathit{Rel}(B)$ is empty. If v is added into $\mathit{Remove}_a(B)$ in some iteration of the while-loop, then $\delta_a(v) \cap \mathit{Rel}(B)$ was nonempty until (B, D) was removed from Rel on line 19 (in the same iteration). Once $\delta_a(v) \cap \mathit{Rel}(B)$ is empty, $\delta_a(v) \cap \mathit{Rel}(B')$ can never get nonempty for any block $B' \subseteq B$ as Rel never grows, and thus v can never be added into some $\mathit{Remove}_a(B')$ for $B' \subseteq B$ on line 23.

The above observation also implies that for a fixed block $B \in P_{\mathit{Sim}_I}$ and $a \in \Sigma$, the sum of all cardinalities of the $\mathit{Remove}_a(B')$ sets, where $B \subseteq B'$ according to which a *Split* is being done, is below $|\delta_a^{-1}(S)|$. Therefore, the overall time taken by splitting on line 10 is in

$$\mathcal{O}\left(\sum_{B \in P_{\mathit{Sim}_I}} \sum_{a \in \mathrm{in}(B)} |\delta_a^{-1}(S)|\right).$$

Complexity of Refinement (lines 16–23). Lines 18 and 19 are $\mathcal{O}(1)$-time (Rel is a boolean matrix). Before we enter the for-loop on line 16, we compute a list $\mathit{RemoveList}_a(B) = \{D \in P \mid D \subseteq \mathit{Remove}\}$. This is an $\mathcal{O}(|\mathit{Remove}|)$ operation and by almost the same argument as in the case of the overall time complexity of *Split* on line 10, we get also exactly the same overall time complexity for computing all the $\mathit{RemoveList}_a(B)$ lists. On line 16, the blocks C are listed by traversing all $\delta^{-1}(v), v \in B$. From the above it follows that for any two $B', D' \in P_{\mathit{Sim}_I}$ and any $a \in \Sigma$, it can happen at most once that a and some B with $B' \subseteq B$ are chosen on line 5 and at the same time $D' \subseteq \mathit{Remove}_a(B)$. Moreover, it holds that $a \in \mathrm{in}(B)$ and $\mathit{Remove}_a(B) \subseteq \delta_a^{-1}(S)$. Thus, for a fixed a, the a-transition leading to a block $B' \in P_{\mathit{Sim}_I}$ can be traversed on line 16 only $|\{D' \in P_{\mathit{Sim}_I} \mid a \in \mathrm{out}(D')\}|$ times and thus the time complexity of lines 16–18 amounts to $\mathcal{O}(\sum_{D \in P_{\mathit{Sim}_I}} \sum_{a \in \mathrm{out}(D)} |\delta_a|)$.

The analysis of lines 19–23 is based on the fact that if some (C, D) appears once on line 19, then no (C', D') with $C' \subseteq C, D' \subseteq D$ can appear there again (as (C, D) is removed from Rel and Rel never grows). Moreover, (C, D) can appear on line 19 only if $C \times D \subseteq \mathit{IO}$. For a fixed (C, D), the time spent in lines 19–23 is in $\mathcal{O}(\sum_{v \in B} |\delta^{-1}(v)|)$ and therefore the overall complexity of lines 19–23 amounts to

$$\mathcal{O}\left(\sum_{B \in P_{\mathit{Sim}_I}} \sum_{v \in \mathit{IO}(B)} |\delta^{-1}(v)|\right).$$

Overall Time Complexity. From the above it follows that the time complexity of OLRT is covered by the following six factors:

1. $\mathcal{O}(|\Sigma||P_{IO}|^2)$,
2. $\mathcal{O}(|\Sigma||S|)$,
3. $\mathcal{O}(|P_{Sim_I}|^2)$,
4. $\mathcal{O}(\sum_{B\in P_{Sim_I}} \sum_{a\in \mathrm{in}(B)} |\delta_a^{-1}(S)|)$,
5. $\mathcal{O}(\sum_{B\in P_{Sim_I}} \sum_{a\in \mathrm{out}(B)} |\delta_a|)$, and
6. $\mathcal{O}(\sum_{B\in P_{Sim_I}} \sum_{v\in IO(B)} |\delta^{-1}(v)|))$.

In total, this gives:

$$\mathcal{O}\left(\begin{array}{c} |\Sigma||P_{IO}|^2 + |\Sigma||S| + |P_{Sim_I}|^2 + \\ \sum\limits_{B\in P_{Sim_I}} \left(\sum\limits_{a\in \mathrm{in}(B)} |\delta_a^{-1}(S)| + \sum\limits_{a\in \mathrm{out}(B)} |\delta_a| + \sum\limits_{v\in IO(B)} |\delta^{-1}(v)| \right) \end{array}\right).$$

Space Comlexity. The space complexity of OLRT is determined by the number of counters, the contents of the *Remove* sets, the size of the matrix encoding of *Rel*, and the Σ-indexed arrays attached to blocks and states. This is covered by the above Factors 2,3 and 4, which gives:

$$\mathcal{O}\left(|P_{Sim_I}|^2 + |\Sigma||S| + \sum_{B\in P_{Sim_I}} \sum_{a\in \mathrm{in}(B)} |\delta_a^{-1}(S)|\right).$$

Observe that the improvement of both time and space complexity of LRT is most significant for systems with large alphabets and a‘high diversity of sets of input and output symbols of states. Certain regular diversity of sets of input and output symbols is an inherent property of LTSs that arise when we compute simulations over tree automata. We address the impact of employing OLRT within the procedures for computing tree automata simulation in the next section.

5.4 Tree Automata Simulations

In [ABH+08], the authors propose methods for computing tree automata simulations via translating problems of computing simulations over tree-automata to problems of computing simulations over certain LTSs. In this section, we show how replacing LRT by OLRT within these translation-based procedures decreases the overall complexity of computing tree-automata simulations. For the rest of this section, we fix a TA $A = (Q, \Sigma, \Delta, F)$. From now on, let D denote the maximal downward simulation on A and U the maximal upward simulation on A induced by D.

To define the translations from downward and upward simulation problems, we need the following notions. Given a transition $t = ((q_1, \ldots, q_n), f, q) \in \Delta$, $(q_1, \ldots, q_n)$ is its *left-hand side* and $t(i) \in (Q \cup \{\Box\})^* \times \Sigma \times Q$ is an *environment*—the tuple which arises from t by replacing state q_i, $1 \leq i \leq n$, at the i^{th} position of the left-hand side of t by the so called hole $\Box \notin Q$. We use *Lhs* of to denote the set of all left-hand sides of A and *Env* to denote the set of all environments of A.

We translate the downward simulation problem on A to the simulation problem on the LTS $A^\bullet = (Q^\bullet, \Sigma^\bullet, \{\delta_a^\bullet \mid a \in \Sigma^\bullet\})$ where $Q^\bullet = \{q^\bullet \mid q \in Q\} \cup \{l^\bullet \mid l \in Lhs\}$, $\Sigma^\bullet = \Sigma \cup \{1, \ldots, r_{\mathsf{m}}\}$, and for each $((q_1, \ldots, q_n), f, q) \in \Delta$, $(q^\bullet, q_1 \ldots q_n^\bullet) \in \delta_f^\bullet$ and $(q_1 \ldots q_n^\bullet, q_i^\bullet) \in \delta_i^\bullet$ for each $i : 1 \leq i \leq n$. The initial relation is simply $I^\bullet = Q^\bullet \times Q^\bullet$. The upward simulation problem is then translated into a simulation problem on LTS $A^\odot = (Q^\odot, \Sigma^\odot, \{\delta_a^\odot \mid a \in \Sigma^\odot\})$, where $Q^\odot = \{q^\odot \mid q \in Q\} \cup \{e^\odot \mid e \in Env\}$, $\Sigma^\odot = \Sigma^\bullet$, and for each $t = ((q_1, \ldots, q_n), f, q) \in \Delta$, for each $1 \leq i \leq n$, $(q_i^\odot, t(i)^\odot) \in \delta_i^\odot$ and $(t(i)^\odot, q^\odot) \in \delta_a^\odot$. The initial relation $I^\odot \subseteq Q^\odot \times Q^\odot$ contains all the pairs $(q^\odot, r^\odot)$ such that $q, r \in Q$ and $r \in F \implies q \in F$, and $((q_1 \ldots q_n, f, q)(i)^\odot, (r_1 \ldots r_n, f, r)(i)^\odot)$ such that $(q_j, r_j) \in D$ for all $j : 1 \leq i \neq j \leq n$. Let $Sim^\bullet$ be the maximal simulation on $A^\bullet$ included in $I^\bullet$ and let $Sim^\odot$ be the maximal simulation on $A^\odot$ included in $I^\odot$. The following theorem shows correctness of the translations.

Theorem 2 ([ABH+08]) *For all $q, r \in Q$, we have $(q^\bullet, r^\bullet) \in Sim^\bullet$ if and only if $(q, r) \in D$ and $(q^\odot, r^\odot) \in Sim^\odot$ if and only if $(q, r) \in U$.*

The states of the LTSs ($A^\bullet$ as well as $A^\odot$) can be classified into several classes according to the sets of input/output symbols. Particularly, $Q^\bullet$ can be classified into the classes $\{q^\bullet \mid q \in Q\}$ and for each $n : 1 \leq n \leq r_{\mathsf{m}}$, $\{q_1 \ldots q_n^\bullet \mid q_1 \ldots q_n \in Lhs\}$, and $Q^\odot$ can be classified into $\{q^\odot \mid q \in Q\}$ and for each $a \in \Sigma$ and $i : 1 \leq i \leq r(a)$, $\{t(i)^\odot \mid t = ((q_1, \ldots, q_n), a, q) \in \Delta\}$. This turns to a significant advantage when computing simulations on $A^\bullet$ or on $A^\odot$ using OLRT instead of LRT. Moreover, we now propose another small optimisation, which is a specialised procedure for computing $\langle P_{I \cap Out} Rel_{I \cap Out} \rangle$ for the both of $A^\odot$, $A^\bullet$. It is based on the simple observation that we need only a constant time (not a time proportional to the size of the alphabet) to determine whether two left-hand sides or two environments are related by the particular *Out* (more specifically, $(e_1^\odot, e_2^\odot) \in Out$ if and only if the inner symbols of e_1 and e_2 are the same, and $(q_1 \ldots q_n^\bullet, r_1 \ldots r_m^\bullet) \in Out$ if and only if $n \leq m$).

5.4.1 Complexity of Computing Simulations over TA

In this section, we only point out the main differences between application of LRT [ABH+08] and OLRT on the LTSs that arise from the translations described above. For implementation details and full complexity analysis of the OLRT versions, see the Section 5.3.2.

To be able to express the complexity of running OLRT on $A^\bullet$ and $A^\odot$, we extend D to the set *Lhs* such that $((q_1 \ldots q_n), (r_1 \ldots r_n)) \in D$ if and only if

$(q_i, r_i) \in D$ for each $i : 1 \leq i \leq n$, and we extend U to the set Env such that

$$((q_1 \dots q_n, f, q)(i), (r_1 \dots r_n, f, r)(i)) \in U$$
$$\iff$$
$$m = n \wedge i = j \wedge (q, r) \in U \wedge (\forall k \in \{1, ..., n\}.\ k \neq i \implies (q_k, r_k) \in D).$$

For a preorder ρ over a set X, we use X/ρ to denote the partition of X according to the equivalence $\rho \cap \rho^{-1}$.

The procedures for computing $Sim^\bullet$ and $Sim^\odot$ consist of (i) translating A to the particular LTS ($A^\bullet$ or $A^\odot$) and computing the partition-relation pair inducing the initial preorder ($I^\bullet$ or $I^\odot$), and (ii) running a simulation algorithm (LRT or OLRT) on it. Below, we analyse the impact of replacing LRT by OLRT on the complexity of step (ii), which is the step with dominating complexity (as shown in [ABH+08] and also by our experiments; step (ii) is much more computationally demanding than step (i)).

We will instantiate the six OLRT time complexity factors for $A^\bullet$ and $A^\odot$. The first time complexity factor comes out from computing $\langle P_{I \cap Out}, Rel_{I \cap Out} \rangle$. For $A^\bullet$ and $A^\odot$, this factor can be decreased exploiting special properties of the transition systems. Bellow, we assume that $|Q| \leq |\Delta|$ and $|Q| \leq |Env|$.

In the case of $A^\bullet$, $\langle P_{I \cap Out}, Rel_{I \cap Out} \rangle$ can be computed separately for the states $Q_1^\bullet = \{q^\bullet \mid q \in Q\}$ and $Q_2^\bullet = \{l^\bullet \mid l \in Lhs\}$. For the first set, for each $a \in \Sigma$, we perform $Split(P, \delta_a^{-1}(Q_2^\bullet))$ and remove from the relation all the pairs of blocks included in $Q_1^\bullet \setminus \delta_a^{-1}(Q_2^\bullet) \times \delta_a^{-1}(Q_2^\bullet)$. Then, we partition $Q_2^\bullet$ into blocks $B_n = \{l^\bullet \mid l = q_1 \dots q_n \in Lhs\}$ for each $n \leq r_\mathsf{m}$ and remove all relation on these blocks $B_n, 1 \leq n \leq r_\mathsf{m}$ apart from the diagonal. Finally, we remove all relations between blocks included in $Q_1^\bullet$ and those in $Q_2^\bullet$. Note that $Q_2^\bullet$ is initially partitioned into r_m blocks (one block for left-hand sides of each possible length up to r_m). Overall, the procedure takes time in $\mathcal{O}(|\Sigma||Q/D|^2 + r_\mathsf{m}^2)$.

In the case of $A^\odot$, $\langle P_{I \cap Out}, Rel_{I \cap Out} \rangle$ is computed separately for the states $Q_1^\odot = \{q^\odot \mid q \in Q\}$ and $Q_2^\odot = \{e^\odot \mid e \in Env\}$. For the first set, for each $i : 1 \leq i \leq r_\mathsf{m}$, we perform $Split(P, \delta_i^{-1}(Q_2^\odot))$ and remove from the relation all pairs of blocks included in $Q_1^\odot \setminus \delta_i^{-1}(Q_2^\odot) \times \delta_i^{-1}(Q_2^\odot)$. Then, for each $a \in \Sigma$, we perform $Split(Q_2^\odot, \delta_a^{-1}(Q_1^\odot))$ and remove from the relation all pairs of blocks included in $Q_2^\odot \setminus \delta_a^{-1}(Q_1^\odot) \times \delta_a^{-1}(Q_1^\odot)$. Finally, we remove all the relations between blocks from $Q_1^\odot$ and $Q_2^\odot$. Overall, the procedure takes time $\mathcal{O}(|\Sigma||Q/U|^2 + |Env/U|^2)$.

We now derive the complexity of computing simulation on $A^\bullet$ by OLRT. The above time complexity factors of OLRT are covered by the following factors (the first factor is the complexity of the specialised procedure for computing $\langle P_{I \cap Out}, Rel_{I \cap Out} \rangle$):

1. $\mathcal{O}(|\Sigma||Q/D|^2 + r_\mathsf{m}^2)$,
2. $\mathcal{O}((r_\mathsf{m} + |\Sigma|)|Lhs \cup Q|)$,
3. $\mathcal{O}(|Lhs \cup Q/D|^2)$,
4. $\mathcal{O}(|\Sigma||Lhs/D||Q| + r_\mathsf{m}|Q/D||Lhs|)$,
5. $\mathcal{O}(r_\mathsf{m}|Lhs/D||Lhs| + |Q/D||\Delta|)$, and

6. $\mathcal{O}(|Lhs/D||\Delta| + r_{\mathsf{m}}|Q/D||Lhs|)$.

The space complexity is the sum of the factors 2–4. This gives $\mathcal{O}(Space_D)$, where $Space_D$ amounts to

$$(r_{\mathsf{m}} + |\Sigma|)|Lhs \cup Q| + |Lhs \cup Q/D|^2 + |\Sigma||Lhs/D||Q| + r_{\mathsf{m}}|Q/D||Lhs|.$$

The time complexity can then be simplified to

$$\mathcal{O}\left(Space_D + |\Sigma||Q/D|^2 + r_{\mathsf{m}}|Lhs/D||Lhs| + |Q/D||\Delta| + |Lhs/D||\Delta|\right).$$

In the case of $A^{\odot}$, the space complexity of running OLRT on $Sim^{\odot}$ and $\langle P_{I^{\odot}}, Rel_{I^{\odot}}\rangle$ is $\mathcal{O}(Space_U)$, where $Space_U = (r_{\mathsf{m}} + |\Sigma|)|Env| + |Env/U|^2 + |Env/U||Q| + |Q/U||Env|$. The space complexity is again the sum of the Factors 2–4 of the six time complexity factors that we instantiate bellow. The first factor is the complexity of the specialised procedure for computing $\langle P_{I\cap Out}, Rel_{I\cap Out}\rangle$. We assume that $Q \leq Env$, and thus also that $Q/U \leq Env/U$:

1. $\mathcal{O}(|\Sigma||Q/U|^2 + |Env/U|^2)$,
2. $\mathcal{O}((r_{\mathsf{m}} + |\Sigma|)|Env|)$,
3. $\mathcal{O}(|Env/U|^2)$,
4. $\mathcal{O}(|Env/U||Q| + |Q/U||Env|)$,
5. $\mathcal{O}(|Env/U||Env| + |Q/U||Env|)$, and
6. $\mathcal{O}(|Env/U||\delta| + |Q/U||Env|)$.

Finally, the time complexity of OLRT can be written as

$$\mathcal{O}(Space_U + |\Sigma||Q/U|^2 + |Env/U||Env| + |Env/U||\delta|).$$

Now, let us compare the above complexities with the results obtained in [ABH+08], where LRT is used. In the case of $A^{\bullet}$ and $I^{\bullet}$, LRT takes $\mathcal{O}(Space_D^{old})$ space where $Space_D^{old} = (|\Sigma| + r_{\mathsf{m}})|Q \cup Lhs||Q \cup Lhs/D|$, and $\mathcal{O}(Space_D^{old} + |\Delta||Q \cup Lhs/D|)$ time. When $A^{\odot}$ and $I^{\odot}$ are considered, we obtain space complexity $\mathcal{O}(Space_U^{old})$ where $Space_U^{old} = |\Sigma||Env||Env/U|$ and time complexity $\mathcal{O}(Space_U^{old} + r_{\mathsf{m}}|\Delta||Env/U|)$.

The biggest difference is in the space complexity (decreasing the factors $Space_D^{old}$ and $Space_U^{old}$). However, the time complexity is better too, and our experiments show a significant improvement in space as well as in time.

5.5 Experimental Results

We implemented the original and the improved version of the algorithm in a uniform way in OCaml and experimentally compared their performance.

The simulation algorithms were benchmarked using LTSs obtained from the runs of the abstract regular model checking (ARMC) (see [BHMV05, BHV04]) on several classic examples—producer-consumer (pc), readers-writers (rw), and

Table 5.1: LTS simulation results

	LTS			LRT		OLRT	
source	$\|S\|$	$\|\Sigma\|$	$\|\delta\|$	time	space	time	space
random	256	16	416	0.12	9.6	0.02	1.9
random	4096	16	3280	13.82	714.2	2.02	78.2
random	16384	16	26208	o.o.m.		268.85	4514.9
random	4096	32	6560	62.09	1844.2	4.36	121.4
random	4096	64	13120	158.38	3763.2	6.59	211.2
pc	1251	43	49076	7.52	418.1	2.63	119.0
rw	4694	11	20452	81.28	3471.8	19.25	989.3
lr	6160	35	90808	390.91	12640.8	45.69	1533.6

Table 5.2: Downward simulation results

	TA				LTS			LRT		OLRT	
source	$\|Q\|$	$\|\Sigma\|$	r_{m}	$\|\Delta\|$	$\|S\|$	$\|\Sigma\|$	$\|\delta\|$	time	space	time	space
random	16	16	2	245	184	18	570	0.06	6.2	0.02	1.4
random	32	16	2	935	655	18	2165	0.87	74.4	0.21	14.4
random	64	16	2	3725	2502	18	8568	26.63	1417.9	3.50	195.4
random	32	32	2	1164	719	34	2511	2.67	166.6	0.23	16.8
random	32	64	2	2026	925	66	3780	12.17	623.5	0.56	25.4
ARTMC[1]	47	132	2	837	241	134	1223	0.84	70.6	0.05	6.2
ARTMC	variable[2]							517.98	116.2	80.84	22.1

list reversal (lr)—and using a set of tree automata obtained from the run of the abstract regular tree model checking (ARTMC) (see [BHH+08]) on several operations, such as list reversal, red-black tree balancing, etc. We also used several randomly generated LTSs and tree automata.

We performed the experiments on AMD Opteron 8389 2.90 GHz PC with 128 GiB of memory (however we set the memory limit to approximately 20 GiB for each process). The system was running Linux and OCaml 3.10.2.

The performance of the algorithms is compared in Table 5.1 (general LTSs), Table 5.2 (LTSs generated while computing the downward simulation), and Table 5.3 (LTSs generated while computing the upward simulation), which contain the running times ([s]) and the amount of memory ([MiB]) required to finish the computation.

As seen from the results of our experiments, our optimised implementation performs substantially better than the original. On average, it improves the running time and space requirements by about one order of magnitude. As expected, we can see the biggest improvements especially in the cases, where we tested the impact of the growing size of the alphabet.

[1] One of the automata selected from the ARTMC set.

[2] A set containing 10 305 tree automata of variable size (up to 50 states and up to 1000 transitions per automaton). The results show the total amount of time required for the computation and the peak size of allocated memory.

Table 5.3: Upward simulation results

	TA				LTS			LRT		OLRT	
source	$\|Q\|$	$\|\Sigma\|$	r_{m}	$\|\Delta\|$	$\|S\|$	$\|\Sigma\|$	$\|\delta\|$	time	space	time	space
random	16	16	2	245	472	17	952	1.03	96.5	0.09	4.8
random	32	16	2	935	1791	17	3700	18.73	1253.8	1.37	54.7
random	64	16	2	3725	7126	17	14824	405.89	14173.9	22.83	752.6
random	32	32	2	1164	2204	33	4548	64.10	3786.7	2.36	193.4
random	32	64	2	2026	3787	65	7874	o.o.m.		6.72	245.8
ARTMC[1]	47	132	2	837	1095	133	3344	66.46	4183.2	0.69	68.2
ARTMC	variable[2]							12669.94	4412.6	400.62	106.6

5.6 Conclusions and Future Work

We have proposed an optimised algorithm for computing simulations over LTSs, which improves the asymptotic complexity in both space and time of the best algorithm (LRT) known to date (see [ABH+08]) and which also performs significantly better in practice. We have also shown how employing OLRT instead of LRT reduces the complexity of the procedures for computing tree automata simulations from [AHKV08]. This is especially important in connection with our algorithms for size reduction and language inclusion presented in Chapter 6 and Chapter 7 and used in the verification procedure proposed in Chapter 4.

As for future work, one can consider further optimisations of the proposed algorithm. Here, an interesting question is whether the impact of the alphabet size to the complexity can further be reduced. One of the possibilities is to represent internal data structures of the OLRT in a more efficient way, for instance, using BDDs.

6 Efficient Inclusion over Tree Automata

As we have already mentioned, finite tree automata play a crucial role in several formal verification techniques, such as (abstract) regular tree model checking [AJMd02, BHRV06a], verification of programs with complex dynamic data structures [BHRV06b], analysis of network firewalls [Bou11], and implementation of decision procedures of logics such as WS2S or MSO [KMS01], which themselves have numerous applications. In the context of verification of programs manipulating dynamically linked data structures, let us also mention (apart from our verification technique presented in Chapter 4, and many others) the work in [MPQ11] which deals with the verification of programs manipulating heap structures with data.

In Chapter 5, we argued that in order to successfully use nondeterministic finite automata, one needs efficient algorithms for handling them. This is notably the case of size reduction and language inclusion that are traditionally done via determinisation. We have already said that determinisation-based size reduction can be replaced by simulation quotienting and we have proposed the algorithm for computing simulations to be used for this purpose. In this chapter, we concentrate on the other problem, i.e., the problem of inclusion checking. For that purpose, algorithms based on using antichains and antichains combined with simulations have been proposed in [BHH$^+$08, ACH$^+$10]. We further improve the state of the art by proposing a new algorithm for inclusion checking that turns out to significantly outperform the existing algorithms in most of our experiments which we performed on two different automata representations. In the first case, the transition function of automata are encoded explicitly, in the second case, the transition function is encoded in a semi-symbolic way using multi-terminal binary decision diagrams (MTBDDs) such that the states stay explicit.

The classic textbook algorithm for checking inclusion $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$ between two TA $\mathcal{A}_S$ (Small) and $\mathcal{A}_B$ (Big) first determinises $\mathcal{A}_B$, computes the complement automaton $\overline{\mathcal{A}_B}$ of $\mathcal{A}_B$, and then checks language emptiness of the product automaton accepting $\mathcal{L}(\mathcal{A}_S) \cap \mathcal{L}(\overline{\mathcal{A}_B})$. This approach has been optimised in [TH03, BHH$^+$08, ACH$^+$10] which describe variants of this algorithm that try to avoid the construction of the whole product automaton (which can be exponentially larger than $\mathcal{A}_B$ and which is indeed extremely large in many practical cases) by constructing some of its states and checking language emptiness on the fly.

By employing the antichain principle within the construction of the product automaton which allows for the given set of states to discard all its supersets, the algorithm is often able to prove or refute inclusion by constructing a small

part of the product automaton only. The work of [TH03] does, in fact, not use the terminology of antichains despite implementing them in a symbolic, BDD-based way. It specialises to binary tree automata only. A more general introduction of antichains within a lattice-theoretic framework appeared in the context of word automata in [DWDHR06]. Subsequently, [BHH+08] has generalised [DWDHR06] for explicit upward inclusion checking on TA and experimentally advocated its use within abstract regular tree model checking.

Additionally, the antichain algorithms can also be combined with using upward simulation relations such as in [ACH+10] (see also [DR10] for other combinations of antichains and simulations for word automata).

Upward Inclusion Checking. In general, we denote the above algorithms for TA as *upward* algorithms to reflect the direction in which they traverse automata $\mathcal{A}_S$ and $\mathcal{A}_B$ (i.e., they start with leave transitions and continue upwards towards the accepting states).

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

Downward Inclusion Checking. The alternative *downward* approach to checking TA language inclusion was first proposed in [HVP05] in the context of subtyping of XML types. This algorithm is not derivable from the textbook approach and has a more complex structure with its own weak points; nevertheless, it does not suffer from the two issues of the upward algorithm mentioned above. We generalise the algorithm of [HVP05] for automata over alphabets with an arbitrary rank ([HVP05] considers rank at most two), and, most importantly, we improve it significantly by using the antichain principle, empowered by a use of the cheap and usually large downward simulation. In this way, we obtain an algorithm which is complementary to and highly competitive with the upward algorithm as shown by our experimental results (in which the newly proposed algorithm significantly dominates in most of the considered cases).

Dealing with Semi-Symbolic Encoding. Certain important applications of TA such as formal verification of programs with complex dynamic data structures or decision procedures of logics such as WS2S or MSO require handling very large alphabets. Here, the common choice is to use the MONA tree automata library [KMS01] which is based on representing transitions of TA symbolically using MTBDDs. However, the encoding used by MONA is restricted to *deterministic* automata only. This implies a necessity of immediate determinisation after each operation over TA that introduces nondeterminism, which very easily leads to a state space explosion. Despite the extensive engineering effort

spent to optimise the implementation of MONA, this fact significantly limits its applicability.

As a way to overcome this difficulty, we have participated on a proposal of a semi-symbolic representation of *nondeterministic* TA which generalises the one used by MONA, and we have developed algorithms implementing the basic operations on TA (such as union, intersection, etc.) as well as more involved algorithms for computing simulations and for checking inclusion (using simulations and antichains to optimise it) over the proposed representation. We have also conducted experiments with a prototype implementation of our algorithms showing again a dominance of downward inclusion checking and justifying usefulness of our symbolic encoding for TA with large alphabets. However, the symbolic encoding is beyond the scope of this work. More details can be found in [HLŠV11a] or in [HLŠV11b]. Here, we only present experimental evaluation in order to compare the inclusion algorithms.

Plan of the Chapter. The rest of this chapter is organised as follows. Section 6.1 describes the newly proposed downward inclusion checking algorithm and several extensions of its basic version which can greatly improve its performance. Section 6.2 provides an experimental evaluation of downward inclusion checking in the explicit and the semi-symbolic setting. Section 6.3 then concludes the chapter.

6.1 Downward Inclusion Checking

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

For instance, given a transition $q \xrightarrow{a} (p_1, p_2)$ in $\mathcal{A}_S$, transitions $r \xrightarrow{a} (s_1, s_2)$ and $r \xrightarrow{a} (t_1, t_2)$ in $\mathcal{A}_B$, and assuming that there are no further top-down transitions from q and r, it holds that $\mathcal{L}(q) \subseteq \mathcal{L}(r)$ if and only if $\mathcal{L}((p_1, p_2)) \subseteq \mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2))$. Note that the union $\mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2))$ cannot be computed component-wise, this is, $\mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2)) \neq (\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times (\mathcal{L}(s_2) \cup \mathcal{L}(t_2))$. For instance, provided $\mathcal{L}(s_1) = \mathcal{L}(s_2) = \{b\}$ and $\mathcal{L}(t_1) = \mathcal{L}(t_2) = \{c\}$, it holds that $\mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2)) = \{(b, b), (c, c)\}$, but the component-wise union is

$$(\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times (\mathcal{L}(s_2) \cup \mathcal{L}(t_2)) = \{(b, b), (b, c), (c, b), (c, c)\}.$$

Hence, we cannot simply check whether $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1)$ and $\mathcal{L}(p_2) \subseteq \mathcal{L}(s_2) \cup \mathcal{L}(t_2)$ to answer the original query, and we have to proceed by checking inclusion on the obtained tuples of states. However, exploring the top-down

transitions that lead from the states that appear in these tuples will lead us to dealing with tuples of tuples of states, etc.

Fortunately, there is a way out of the above trap. In particular, as first observed in [HVP05] in the context of XML type checking, we can exploit the following property of the Cartesian product of sets $G, H \subseteq \mathcal{U}$:

$$G \times H = (G \times \mathcal{U}) \cap (\mathcal{U} \times H). \tag{6.1}$$

Continuing in our example, we can rewrite

$$\begin{aligned} \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq (\mathcal{L}((s_1, s_2)) \cup \mathcal{L}((t_1, t_2))) = \\ (\mathcal{L}(s_1) \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times \mathcal{L}(t_2)) \end{aligned} \tag{6.2}$$

as

$$\begin{aligned} \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cap (T_\Sigma \times \mathcal{L}(s_2))) \cup \\ ((\mathcal{L}(t_1) \times T_\Sigma) \cap (T_\Sigma \times \mathcal{L}(t_2))), \end{aligned} \tag{6.3}$$

This can further be rewritten, using the distributive laws in the $(2^{T_\Sigma \times T_\Sigma}, \subseteq)$ lattice, as

$$\begin{aligned} \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \cap \\ ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2))) \cap \\ ((T_\Sigma \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \cap \\ ((T_\Sigma \times \mathcal{L}(s_2)) \cup (T_\Sigma \times \mathcal{L}(t_2))). \end{aligned} \tag{6.4}$$

It is easy to see that the inclusion holds exactly if it holds for all components of the intersection, i.e., if and only if

$$\begin{aligned} &\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \wedge \\ &\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2))) \wedge \\ &\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((T_\Sigma \times \mathcal{L}(s_2)) \cup (\mathcal{L}(t_1) \times T_\Sigma)) \wedge \\ &\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((T_\Sigma \times \mathcal{L}(s_2)) \cup (T_\Sigma \times \mathcal{L}(t_2))). \end{aligned} \tag{6.5}$$

Two things should be noted in the previous equation.

1. If we are computing the union of languages of two tuples such that they have T_Σ at all indices other than some index i, we can compute it component-wise, i.e.,

$$\begin{aligned} \mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (\mathcal{L}(t_1) \times T_\Sigma)) = \\ (\mathcal{L}(s_1) \cup \mathcal{L}(t_1)) \times T_\Sigma. \end{aligned} \tag{6.6}$$

   The above clearly holds iff $\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1)$.

2. If T_Σ does not appear at the same positions as in the inclusion

$$\mathcal{L}(p_1) \times \mathcal{L}(p_2) \subseteq ((\mathcal{L}(s_1) \times T_\Sigma) \cup (T_\Sigma \times \mathcal{L}(t_2))), \tag{6.7}$$

   it must hold that either

$$\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \qquad \text{or} \qquad \mathcal{L}(p_2) \subseteq \mathcal{L}(t_2). \tag{6.8}$$

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

$$\begin{aligned} \mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \cup \mathcal{L}(t_1) \quad &\wedge \\ (\mathcal{L}(p_1) \subseteq \mathcal{L}(s_1) \vee \mathcal{L}(p_2) \subseteq \mathcal{L}(t_2)) \quad &\wedge \\ (\mathcal{L}(p_1) \subseteq \mathcal{L}(t_1) \vee \mathcal{L}(p_2) \subseteq \mathcal{L}(s_2)) \quad &\wedge \\ \mathcal{L}(p_2) \subseteq \mathcal{L}(s_2) \cup \mathcal{L}(t_2). \end{aligned} \tag{6.9}$$

The above reasoning can be generalised to dealing with transitions of any arity as shown in Theorem 3. In the theorem, we conveniently exploit the notion of *choice functions*. Given $P_B \subseteq Q_B$ and $a \in \Sigma$, $\#a = n \geq 1$, we denote by $cf(P_B, a)$ the set of all choice functions f that assign an index i, $1 \leq i \leq n$, to all n-tuples $(q_1, \ldots, q_n) \in Q_B^n$ such that there exists a state in P_B that can make a transition over a to $(q_1, \ldots, q_n)$; formally, $cf(P_B, a) = \{f : down_a(P_B) \to \{1, \ldots, \#a\}\}$.

Theorem 3 *Let $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$ be tree automata. For sets $P_S \subseteq Q_S$ and $P_B \subseteq Q_B$ it holds that $\mathcal{L}(P_S) \subseteq \mathcal{L}(P_B)$ if and only if $\forall p_S \in P_S \; \forall a \in \Sigma$: if $p_S \xrightarrow{a} (r_1, \ldots, r_{\#a})$, then it holds that $down_a(P_B) = \{()\}$ when $\#a = 0$ and*

$$\forall f \in cf(P_B, a) \; \exists 1 \leq i \leq \#a : \mathcal{L}(r_i) \subseteq \bigcup_{\substack{(u_1, \ldots, u_{\#a}) \in down_a(P_B) \\ f((u_1, \ldots, u_{\#a})) = i}} \mathcal{L}(u_i)$$

when $\#a > 0$.

Proof. For two sets $P_S \subseteq Q_S, P_B \subseteq Q_B$, it clearly holds that $\mathcal{L}(P_S) \subseteq \mathcal{L}(P_B)$ if and only if $\forall p_S \in P_S \; \forall a \in \Sigma$:

$$p_S \xrightarrow{a} (r_1, \ldots, r_n) \implies \mathcal{L}((r_1, \ldots, r_n)) \subseteq \bigcup_{\overline{u} \in down_a(P_B)} \mathcal{L}((u_1, \ldots, u_n)) \tag{6.10}$$

where $\overline{u} = (u_1, \ldots, u_n)$. For the case when $\#a = 0$, the above formula collapses to $p_S \xrightarrow{a} () \implies \mathcal{L}(()) \subseteq \cup_{() \in down_a(P_B)} \mathcal{L}(())$. Since $down_a(P_B) \subseteq \{()\}$ for $\#a = 0$, the first part of the theorem is proven. We prove the second part (when $\#a > 0$) in the following steps. Let us fix $n = \#a$, $\overline{u} = u_1, \ldots, u_n$:

$$\begin{aligned} \mathcal{L}((r_1, \ldots, r_n)) \subseteq \bigcup_{\overline{u} \in down_a(P_B)} \mathcal{L}((u_1, \ldots, u_n)) \iff \\ \prod_{i=1}^{n} \mathcal{L}(r_i) \subseteq \bigcup_{\overline{u} \in down_a(P_B)} \prod_{i=1}^{n} \mathcal{L}(u_i), \end{aligned} \tag{6.11}$$

where $\prod_{i=1}^{n} S_i$ denotes the Cartesian product of a family of sets $\{S_1, \ldots, S_n\}$. We can further observe that for a set $\mathcal{U}$ and a family of sets $\{S_1, \ldots, S_n\} : \forall 1 \leq i \leq n : S_i \subseteq \mathcal{U}$ it holds that

$$\prod_{i=1}^{n} S_i = \bigcap_{i=1}^{n} \mathcal{U}^{i-1} \times S_i \times \mathcal{U}^{n-i}. \tag{6.12}$$

Given the family of sets $\{\mathcal{L}(u_1), \ldots, \mathcal{L}(u_n)\}$ and the decomposition from Equation 6.12 we can modify the formula from Equation 6.11 in the following way:

$$\begin{aligned} \prod_{i=1}^{n} \mathcal{L}(r_i) \subseteq & \bigcup_{\overline{u} \in down_a(P_B)} \prod_{i=1}^{n} \mathcal{L}(u_i) & \iff \\ \prod_{i=1}^{n} \mathcal{L}(r_i) \subseteq & \bigcup_{\overline{u} \in down_a(P_B)} \bigcap_{i=1}^{n} T_\Sigma^{i-1} \times \mathcal{L}(u_i) \times T_\Sigma^{n-i}. & \end{aligned} \tag{6.13}$$

Since the power set lattice $(2^{Q_B}, \subseteq)$ is a completely distributive lattice, we can exploit the fact that for any doubly indexed family $\{x_{j,k} \in 2^{Q_B} \mid j \in J, k \in K_j\}$ it holds that

$$\bigcup_{j \in J} \bigcap_{k \in K_j} x_{j,k} = \bigcap_{f \in F} \bigcup_{j \in J} x_{j,f(j)} \tag{6.14}$$

where F is the set of choice functions f choosing for each index $j \in J$ some index $f(j) \in K_j$. For our purpose, we introduce the set of choice functions:

$$F = \{f : down_a(P_B) \to \{1, \ldots, n\}\}. \tag{6.15}$$

Therefore,

$$\begin{aligned} \prod_{i=1}^{n} \mathcal{L}(r_i) \subseteq & \bigcup_{\overline{u} \in down_a(P_B)} \bigcap_{i=1}^{n} T_\Sigma^{i-1} \times \mathcal{L}(u_i) \times T_\Sigma^{n-i} & \iff \\ \prod_{i=1}^{n} \mathcal{L}(r_i) \subseteq & \bigcap_{f \in F} \bigcup_{\overline{u} \in down_a(P_B)} T_\Sigma^{f(\overline{u})-1} \times \mathcal{L}(u_{f(\overline{u})}) \times T_\Sigma^{n-f(\overline{u})}. & \end{aligned} \tag{6.16}$$

Due to the fact that for a set S, its subset $T \subseteq S$ and a family of its subsets $R \subseteq 2^S$ it holds that

$$T \subseteq \bigcap_{U \in R} U \iff \forall U \in R : T \subseteq U, \tag{6.17}$$

we can simplify our case in the following way:

$$\begin{aligned} \prod_{i=1}^{n} \mathcal{L}(r_i) \subseteq & \bigcap_{f \in F} \bigcup_{\overline{u} \in down_a(P_B)} T_\Sigma^{f(\overline{u})-1} \times \mathcal{L}(u_{f(\overline{u})}) \times T_\Sigma^{n-f(\overline{u})} \\ & \iff \\ \forall f \in F : \prod_{i=1}^{n} \mathcal{L}(r_i) \subseteq & \bigcup_{\overline{u} \in down_a(P_B)} T_\Sigma^{f(\overline{u})-1} \times \mathcal{L}(u_{f(\overline{u})}) \times T_\Sigma^{n-f(\overline{u})} \end{aligned} \tag{6.18}$$

For some fixed f, we can further rewrite the right-hand side of the inclusion query to the following:

$$\begin{aligned} & \bigcup_{\overline{u} \in down_a(P_B)} T_\Sigma^{f(\overline{u})-1} \times \mathcal{L}(u_{f(\overline{u})}) \times T_\Sigma^{n-f(\overline{u})} & = \\ & \bigcup_{i=1}^{n} \bigcup_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u})=i}} T_\Sigma^{i-1} \times \mathcal{L}(u_i) \times T_\Sigma^{n-i} & = \\ & \bigcup_{i=1}^{n} \left[T_\Sigma^{i-1} \times \left[\bigcup_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u})=i}} \mathcal{L}(u_i) \right] \times T_\Sigma^{n-i} \right] & \end{aligned} \tag{6.19}$$

It can be observed that for a set $\mathcal{U}$ and two families of sets $\{S_1, \ldots, S_n\}$ and $\{S'_1, \ldots, S'_n\}$ such that $\forall 1 \leq i \leq n : S_i, S'_i \subseteq \mathcal{U}$ it holds that

$$\prod_{i=1}^{n} S_i \subseteq \bigcup_{i=1}^{n} \mathcal{U}^{i-1} \times S'_i \times \mathcal{U}^{n-i} \iff \exists 1 \leq i \leq n : S_i \subseteq S'_i. \tag{6.20}$$

Hence, we can now finally deduce that

$$\begin{aligned} \forall f \in F : \prod_{i=1}^{n} \mathcal{L}(r_i) \subseteq \bigcup_{i=1}^{n} \left[T_\Sigma^{i-1} \times \left[\bigcup_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u})=i}} \mathcal{L}(u_i) \right] \times T_\Sigma^{n-i} \right] \\ \iff \\ \forall f \in F \; \exists 1 \leq i \leq n : \mathcal{L}(r_i) \subseteq \bigcup_{\substack{\overline{u} \in down_a(P_B) \\ f(\overline{u})=i}} \mathcal{L}(u_i), \end{aligned} \tag{6.21}$$

which concludes the proof. □

6.1.1 Basic Algorithm

We now construct a basic algorithm for downward inclusion checking on tree automata $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$. The algorithm is shown as Algorithm 4. Its main idea relies on a recursive application of Theorem 3 in function `expand1`. The function is given a pair $(p_S, P_B) \in Q_S \times 2^{Q_B}$ for which we want to prove that $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$—initially, the function is called for every pair (q_S, F_B) where $q_S \in F_S$. The function enumerates all possible top-down transitions that $\mathcal{A}_S$ can do from p_S (lines 3–8). For each such transition, the function either checks whether there is some transition $p_B \xrightarrow{a}$ for $p_B \in P_B$ if $\#a = 0$ (line 5), or it starts enumerating and recursively checking queries $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ on which the result of $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ depends according to Theorem 3 (lines 9–16).

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

Using Theorem 3 and noting that Algorithm 4 necessarily terminates because all its loops are bounded, and the recursion in function `expand1` is also bounded due to the use of *workset*, it is not difficult to see that the following theorem holds.

Theorem 4 *Let $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$ be two TA. Algorithm 4 terminates and returns true if and only if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$.*

Algorithm 4: Downward inclusion

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

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

Function expand1(p_S, P_B, *workset*)

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

6.1.2 Antichains and Simulation

In this section, we propose several optimisations of the basic algorithm presented above that, according to our experiments, often have a huge impact on the efficiency of the algorithm—making it in many cases the most efficient algorithm for checking inclusion on tree automata that we are currently aware of. In general, the optimisations are based on an original use of simulations and antichains in a way suitable for the context of downward inclusion checking.

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

1. If there exists a state $p_B \in P_B$ such that $p_S \preceq p_B$, then the inclusion clearly holds (from the assumption made about $\preceq$), and no further checking is needed.

2. Next, it can be seen without any further computation that the inclusion does *not* hold if there exists some (p'_S, P'_B) such that $p'_S \preceq p_S$ and $P_B \preceq^{\forall\exists}$

Algorithm 5: Downward inclusion (antichains + preorder)

Input: TA $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S), \mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B), \preceq \subseteq (Q_S \cup Q_B)^2$
Output: *true* if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$, *false* otherwise
Data: $NN := \emptyset$

```
1 foreach q_S ∈ F_S do
2     if ¬expand2(q_S, F_B, ∅) then return false;
3 return true;
```

Function expand2(p_S, P_B, *workset*)

```
    /* p_S ∈ Q_S , P_B ⊆ Q_B , and workset ⊆ Q_S × 2^{Q_B} */
1  if ∃(p'_S, P'_B) ∈ workset : p_S ⪯ p'_S ∧ P'_B ⪯^{∀∃} P_B then return true;
2  if ∃(p'_S, P'_B) ∈ NN : p'_S ⪯ p_S ∧ P_B ⪯^{∀∃} P'_B then return false ;
3  if ∃p ∈ P_B : p_S ⪯ p then return true;
4  workset := workset ∪ {(p_S, P_B)};
5  foreach a ∈ Σ do
6      if #a = 0 then
7          if down_a(p_S) ≠ ∅ ∧ down_a(P_B) = ∅ then return false;
8      else
9          W := down_a(P_B);
10         foreach (r_1, ..., r_{#a}) ∈ down_a(p_S) do        /* p_S -a-> (r_1, ..., r_{#a}) */
11             foreach f ∈ {W → {1, ..., #a}} do
12                 found := false;
13                 foreach 1 ≤ i ≤ #a do
14                     S := {q_i | (q_1, ..., q_{#a}) ∈ W, f((q_1, ..., q_{#a})) = i}^⪯;
15                     if expand2(r_i, S, workset) then
16                         found := true;
17                         break;
18                     if ∄(r', H) ∈ NN : r' ⪯ r_i ∧ S ⪯^{∀∃} H then
19                         NN := (NN \ {(r', H) | H ⪯^{∀∃} S, r_i ⪯ r'}) ∪ {(r_i, S)};
20                 if ¬found then return false;
21 return true;
```

P'_B, and we have already established that $\mathcal{L}(p'_S) \not\subseteq \mathcal{L}(P'_B)$. Indeed, we have $\mathcal{L}(P_B) \subseteq \mathcal{L}(P'_B) \not\supseteq \mathcal{L}(p'_S) \subseteq \mathcal{L}(p_S)$, and therefore $\mathcal{L}(p_S) \not\subseteq \mathcal{L}(P_B)$.

3. Finally, we can stop evaluating the given inclusion query if there is some $(p'_S, P'_B) \in$ *workset* such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$. Indeed, this means that the result of $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ depends on the result of $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$. However, if $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ holds, then also $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ holds because we have $\mathcal{L}(p_S) \subseteq \mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B) \subseteq \mathcal{L}(P_B)$. On the other hand, if $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ does *not* hold, the path between (p'_S, P'_B) and (p_S, P_B) cannot be the only reason for that since a counterexample has not been found on that path yet, and the chance of finding a counterexample is only smaller from (p_S, P_B).

The version of Algorithm 4 including all the above proposed optimisations is shown as Algorithm 5. The optimisations can be found in the function `expand2` that replaces the function `expand1`. In particular, line 3 implements the first optimisation, line 2 the second one, and line 1 the third one. In order to

implement the second optimisation, the algorithm maintains a new set NN. This set stores pairs (p_S, P_B) for which it has already been shown that the inclusion $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ does not hold.

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

Taking into account Theorem 4 and the above presented facts, it is not difficult to see that the following holds.

Theorem 5 *Let $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$ be two TA. Algorithm 5 terminates and returns true if and only if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$.*

6.1.3 Caching Positive Pairs

The inclusion checking presented in the previous section can be optimised even more. Recall that the algorithm caches pairs for which the inclusion does not hold, i.e., pairs (p_S, P_B) such that $\mathcal{L}(p_S) \not\subseteq \mathcal{L}(P_B)$, in the set NN (which is maintained as an antichain). A natural question that arises is whether there is a similar option for pairs for which the inclusion does hold, i.e., pairs (p_S, P_B) such that $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$. Such an option indeed exists and is presented in the rest of this subsection.

Let us denote the set of the above-mentioned pairs for which the inclusion holds as IN. When checking the inclusion $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$, we first try to find a pair $(p'_S, P'_B) \in IN$ such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$. If such pair exists, then we immediately know that the checked inclusion holds because $\mathcal{L}(p_S) \subseteq \mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B) \subseteq \mathcal{L}(P_B)$.

The set IN can again be optimised as an antichain but with the opposite ordering than NN. This means that there are no two pairs $(p_S, P_B), (p'_S, P'_B)$ such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$ in IN. It is easy to understand that a pair (p_S, P_B) does not have to be stored since whenever (p_S, P_B) can be used to prune the search, (p'_S, P'_B) can also be used.

However, adding new pairs to IN is not as straightforward as for NN. Assume that we add a pair (p_S, P_B) to IN immediately when the function call `expand2`$(p_S, P_B, \mathit{workset})$ at line 15 of Algorithm 5 returns *true* for some *workset*. This is not correct as shown in the following example.

Suppose that when checking inclusion $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$, a test for inclusion $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ where $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$ is encountered somewhere deep in the recursive calls of `expand2`. As stated previously, the inclusion $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ does not need to be tested since if $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$, then $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$, and if $\mathcal{L}(p'_S) \not\subseteq \mathcal{L}(P'_B)$, then this cannot be caused solely by $\mathcal{L}(p_S) \not\subseteq \mathcal{L}(P_B)$. Hence, `expand2`$(p_S, P_B, \mathit{workset})$ returns *true*, and the result of the query $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ will be given by other branches of the call tree generated for the $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ query. However, if we put the pair (p_S, P_B) into IN and later proved that $\mathcal{L}(p'_S) \not\subseteq \mathcal{L}(P'_B)$, then the set IN would become invalid.

Algorithm 6: Downward inclusion (antichains + preorder + IN)

Input: TA $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S), \mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B), \preceq \subseteq (Q_S \cup Q_B)^2$
Output: *true* if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$, *false* otherwise
Data: $IN := NN := \emptyset$

```
1  foreach q_S ∈ F_S do
2      if expand2e(q_S, F_B, ∅) = (false, _, _) then return false;
3  return true;
```

Function expand2e(p_S, P_B, *workset*)

```
   /* p_S ∈ Q_S , P_B ⊆ Q_B , and workset ⊆ Q_S × 2^{Q_B} */
1  if ∃(p'_S, P'_B) ∈ IN : p_S ⪯ p'_S ∧ P'_B ⪯^{∀∃} P_B then return (true, ∅, ∅);
2  if ∃(p'_S, P'_B) ∈ NN : p'_S ⪯ p_S ∧ P_B ⪯^{∀∃} P'_B then return (false, ∅, ∅);
3  if ∃p ∈ P_B : p_S ⪯ p then return (true, ∅, ∅);
4  if ∃(p'_S, P'_B) ∈ workset : p_S ⪯ p'_S ∧ P'_B ⪯^{∀∃} P_B then return
   (true, {(p'_S, P'_B)}, ∅);
5  workset := workset ∪ {(p_S, P_B)};
6  Ant := Con := ∅;
7  foreach a ∈ Σ do
8      if #a = 0 then
9          if down_a(p_S) ≠ ∅ ∧ down_a(P_B) = ∅ then return (false, ∅, ∅);
10     else
11         W := down_a(P_B);
12         foreach (r_1, ..., r_{#a}) ∈ down_a(p_S) do        /* p_S -a-> (r_1, ..., r_{#a}) */
13             foreach f ∈ {W → {1, ..., #a}} do
14                 found := false;
15                 foreach 1 ≤ i ≤ #a do
16                     S := {q_i | (q_1, ..., q_{#a}) ∈ W, f((q_1, ..., q_{#a})) = i}^⪯;
17                     (x, Ant', Con') := expand2e(r_i, S, workset);
18                     if x then
19                         found := true; Ant := Ant ∪ Ant'; Con := Con ∪ Con';
20                         break;
21                     if ∄(r', H) ∈ NN : r' ⪯ r_i ∧ S ⪯^{∀∃} H then
22                         NN := (NN \ {(r', H) | H ⪯^{∀∃} S, r_i ⪯ r'}) ∪ {(r_i, S)};
23                 if ¬found then return (false, ∅, ∅);
24 Ant := Ant \ {(p_S, P_B)}; Con := Con ∪ {(p_S, P_B)};
25 if Ant = ∅ then
26     foreach (x, Y) ∈ Con do
27         if ∄(p'_S, P'_B) ∈ IN : x ⪯ p'_S ∧ P'_B ⪯^{∀∃} Y then
28             IN := (IN \ {(r', H) | Y ⪯^{∀∃} H, r' ⪯ x}) ∪ {(x, Y)};
29     Con := ∅;
30 return (true, Ant, Con);
```

A solution to this issue is given in Algorithm 6. The `expand2e` function is a modified version of `expand2` that additionally returns a formula of the form $\bigwedge Ant \to \bigwedge Con$ where Con (*consequents*) is a set of inclusion queries that can be answered positively provided that the inclusion queries in Ant (*antecedents*) are all answered positively.

When the recursive call of `expand2e`$(p_S, P_B, workset)$ is at the bottom of the call tree and there is $(p'_S, P'_B) \in workset$ such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$

(line 4), then according to the above, the formula returned from `expand2e` along with *true* could be $\bigwedge\{\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)\} \rightarrow \bigwedge\{\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)\}$ because $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ cannot be considered guaranteed before $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ is positively answered. This formula is, however, simplified to $\bigwedge\{\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)\} \rightarrow \emptyset$ since $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ can be forgotten as it is weaker than $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$.

A situation similar to what we have just discussed arises when the recursive call of `expand2e`$(p_S, P_B, \mathit{workset})$ is at the bottom of the call tree and there is $(p'_S, P'_B) \in IN$ such that $p_S \preceq p'_S$ and $P'_B \preceq^{\forall\exists} P_B$ (line 1). In this case, $\bigwedge \emptyset \rightarrow \bigwedge \emptyset$ is returned (along with *true*) since the validity of $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ has already been established. Next, if the recursive call of `expand2e`$(p_S, P_B, \mathit{workset})$ is at the bottom of the call tree and there is $p \in P_B$ such that $p_S \preceq p$ (line 3), $\bigwedge \emptyset \rightarrow \bigwedge \emptyset$ is again returned since for any inclusion query $\mathcal{L}(p'_S) \subseteq \mathcal{L}(P'_B)$ such that $p'_S \preceq p_S$ and $P_B \preceq^{\forall\exists} P'_B$, it will be the case that there is $p' \in P'_B$ such that $p'_S \preceq p'$ (and hence the computation will be immediately stopped without a need to use IN for this purpose). Finally, when `expand2e` returns *false* (line 2), it is accompanied by the formula $\bigwedge \emptyset \rightarrow \bigwedge \emptyset$, which, however, is not taken into account in this case and is returned just to make the result of `expand2e` to have the same structure.

For inner nodes of the call tree, this is, nodes that correspond to function calls `expand2e`(p_S, P_B) that themselves call `expand2e`, all antecedents and consequents returned from successful nested calls are collected into sets *Ant* and *Con*. Then, the condition $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ is removed from *Ant* (if it is there) and added to *Con* since it has just been proved that $\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)$ holds provided that the elements from $\mathit{Ant} \setminus \{\mathcal{L}(p_S) \subseteq \mathcal{L}(P_B)\}$ are later proved to also hold. When the set *Ant* becomes empty, yielding the formula $\bigwedge \emptyset \rightarrow \bigwedge \mathit{Con}$, all elements of *Con* can be added to IN (while respecting the antichain property of IN) and the set *Con* cleared.

Taking into account Theorem 5 and the above presented facts, it can be seen that the following holds.

Theorem 6 *Let $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$ and $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$ be two TA. Algorithm 6 terminates and returns true if and only if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$.*

6.2 Experimental Results

We have experimented with two tree automata encodings. The first one represents the automata explicitly, i.e., the transition function is stored as a plain list. The second encoding stores the automata in a semi-symbolic way using MTBDDs allowing for a convenient representation of automata with large alphabets.

6.2.1 Explicit Encoding

We have implemented three versions of downward inclusion for exlicitly encoded tree automata. Algorithm 5 using an identity as an input preorder is denoted by `down`, Algorithm 5 using the maximum downward simulation as the input preorder is marked as `down+s`. Algorithm 6 is denoted by `down+opt` when

Table 6.1: A comparison of inclusion checking methods on explicit encoding

	small TA (timeout 20 s)			big TA (timeout 60 s)		
	winner	winner (wo sim)	timeout	winner	winner (wo sim)	timeout
`up`	25.52 %	-	0.00 %	3.13 %	-	48.44 %
`up+s`	0.00 %	81.82 %	0.00 %	0.00 %	20.31%	48.44 %
`down`	16.21 %	-	25.08 %	39.06 %	-	60.94 %
`down+s`	3.45 %	18.18 %	0.93 %	14.06 %	79.69%	9.38 %
`down+opt`	42.07 %	-	25.08 %	0.00 %	-	60.94 %
`down+opt+s`	16.76 %	0.00 %	0.62 %	56.25 %	0.00%	1.56 %

parametrised by identity. `down+opt+s` denotes the combination with maximum downward simulation. We have also implemented the algorithm of upward inclusion checking using antichains from [BHH$^+$08] and its modification using upward simulation proposed in [ACH$^+$10] (these algorithms are marked as `up` and `up+s` below). We tested our approach on 387 tree automata pairs of different sizes generated from the intermediate steps of abstract regular tree model checking of the algorithm for rebalancing red-black trees after insertion or deletion of a leaf node [BHH$^+$08]. The automata have been divided into two sets depending on their size. The first set (denoted by "small TA") contains 323 automata of size ranging between 50 and 250 states. The second set (denoted by "big TA") contains 64 automata of size ranging between 400 and 600 states.

The results of the experiments with explicitly encoded automata are presented in Table 6.1 which compares the methods according to the percentage of the cases in which they were the fastest when checking inclusion on the same automata pair. The results are grouped into two sets according to the size of the automata measured in the number of states. Columns "winner (wo sim)" contain the percentage of the cases in which the method was the fastest without counting the time for computing simulation (in such cases they were always faster than the methods not using simulations). This comparison is motivated by the observation that inclusion checking may be used as a part of a bigger computation that anyway computes the simulation relations (which happens, e.g., in abstract regular model checking where the simulations are used for reducing the size of the encountered automata). Finally, the columns labelled by "timeout" summarise how often the particular methods exceeded the time available for the computation of a single inclusion query.

The results show that the overhead of computing upward simulation is too high in all the cases that we have considered, causing upward inclusion checking using simulation to be the slowest when the time for computing the simulation used by the algorithm is included[1]. Next, it can be seen that for each of the remaining approaches there are cases in which they win in a significant way. However, the downward approaches are clearly dominating in significantly more of our test cases (with the only exception being the case of small automata when the time of computing simulations is not included). Moreover, the dominance

[1]Note that `up+s` was winning over `up` in the experiments of [ACH$^+$10] even with the time for computing simulation included, which seems to be caused by a much less efficient implementation of the antichains in the original algorithm.

Table 6.2: A comparison of inclusion checking methods on semi-symbolic encoding

	winner	timeout
`symup`	6.67 %	22.68 %
`symdown`	90.67 %	22.68 %
`symdown+s`	2.67 %	73.92 %

of the downward checking increases with the size of the automata that we considered in our test cases. Finally, one can observe that the `down+opt+s` can solve almost all queries within the given timeout on both sets of automata even though it does not dominate on the set of smaller automata performance wise.

6.2.2 Semi-symbolic Encoding

In addition to experiments with explicitly encoded automata, we have also benchmarked our algorithms on automata represented in a semi-symbolic way using MTBDDs for storing transition functions. We have implemented some of the above mentioned algorithms on top of the MTBDD implementation found in the CUDD library [Som11]. A detailed description of the algorithms modified for the semi-symbolic setting can be found in [HLŠV11b].

We have compared the upward inclusion checking algorithm from [BHH+08] adapted for semi-symbolically represented tree automata, which is given in [HLŠV11b] (and marked as `symup` in the following), with the downward inclusion checking algorithm presented above. In the latter case, we let the algorithm use either the identity relation, which corresponds to downward inclusion checking without using any simulation (this case is marked as `symdown` below), or the maximum downward simulation (which is marked as `symdown+s` in the results). We have not considered upward inclusion checking with upward simulation due to its negative results in our experiments with explicitly encoded automata[2]. The implementation of downward inclusion checking with caching of positive pairs on top of the MTBDD package found in CUDD is also not available (see Chapter 7 for the version built on top of our own MTBDD library). For the comparison, we used 97 pairs of tree automata with a large alphabet which we encoded into 12 bits. The size of the automata was between 50 and 150 states and the timeout was set to 300 s. The automata were obtained by taking the automata considered in Section 6.2.1 and labelling their transitions by randomly generated sets of symbols from the considered large alphabet.

The results that we have obtained are presented in Table 6.2. The first column compares the methods according to the percentage of the cases in which they were the fastest when checking inclusion on the same automata pair. The second column summarises how often each of the methods exceeded the time available for computation.

When we compare the above experimental results with the results obtained on the explicitly represented automata presented in Section 6.2.1, we may note that

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

(1) downward inclusion checking is again significantly dominating, but (2) the advantage of exploiting downward simulation has decreased. According to the information we gathered from code profiling of our implementation, this is due to the overhead of the CUDD library which is used as the underlying layer for storage of shared MTBDDs of several data structures. This has indicated a need of a different MTBDD library to be used or perhaps of a specialised MTBDD library to be developed. The issue is further briefly discussed in Chapter 7.

Finally, we also evaluated performance of the implementation of the described algorithms using a semi-symbolic encoding of TA with performance of the algorithms using an explicit encoding of TA considered in Section 6.1 on the automata with the large alphabet. As we have expected, the symbolic version was on average about 8000 times faster than the explicit one in this case.

6.3 Conclusions and Future Work

We have proposed a new algorithm for checking language inclusion over nondeterministic TA (based on the one from [HVP05]) that traverses automata in a downward manner and uses both antichains and simulations to optimise its computation. This algorithm is, according to our experimental results, mostly superior to the known upward algorithms. We have further briefly presented a semi symbolic MTBDD-based representation of nondeterministic TA generalising the one used by MONA. We have experimentally justified usefulness of the semi-symbolic encoding for nondeterministic TA with large alphabets.

Furthermore, our experiments with downward inclusion show that the performance of the algorithm is heavily dependent on the sequence in which one evaluates successors of pairs of states. In the future, it would be interesting to explore whether a more efficient order of exploring these successors exists.

7 A Tree Automata Library

This chapter describes the general-purpose tree automata library that was designed within our research. The library contains an efficient implementation of many important algorithms for use of TA in symbolic verification such as the algorithm for computing simulations over TA presented in Chapter 5 or various inclusion checking methods described in [ACH+10] or in Chapter 6.

The main motivation behind the development of the library is to achieve a better performance of the Forester verification tool presented in Chapter 4. Apart from this tool, as we have already discussedthere are other formal verification techniques relying on finite tree automata which often strongly depends on the performance of the underlying implementation of TA.

Currently, there exist several available tree automata libraries, which are mostly written in OCaml (e.g., Timbuk/Taml [Gen03]) or Java (e.g., LETHAL [CJH+09]), and they do not always use the most advanced algorithms known to date. Therefore, they are not suitable for tasks which require that the available processing power is utilised as efficiently as possible. An exception from these libraries is MONA [KMS01] implementing decision procedures over WS1S/WS2S, which contains a highly optimised TA package written in C, but, alas, it supports only binary deterministic tree automata. As we have already mentioned, the determinisation is often a very significant bottleneck, and a lot of effort has therefore been invested into developing efficient algorithms for handling nondeterministic tree automata without a need to ever determinise them.

In order to allow researchers focus on developing verification techniques rather than reimplementing and optimising a TA package, we provide VATA[1], an easy-to-use open-source library for efficient manipulation of nondeterministic TA. VATA supports many of the operations commonly used in automata-based formal verification techniques over two complementary encodings: explicit and semi-symbolic. The *explicit* encoding is suitable for most applications that do not need to use alphabets with a large number of symbols. However, some formal verification approaches make use of such alphabets, e.g., the approach for verification of programs with complex dynamic data structures [BHRV06a] or decision procedures of the MSO or WSkS logics [KMS01]. Therefore, in order to address this issue, we also provide a *semi-symbolic* encoding of TA, which uses *multi-terminal binary decision diagrams* [CMZ+97] (MTBDDs), an extension of reduced ordered binary decision diagrams [Bry86] (BDDs), to store the transition function of TA. In order to enable the widest possible range of applications of the library even for the semi-symbolic encoding, we provide both bottom-up and top-down semi-symbolic representations.

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`

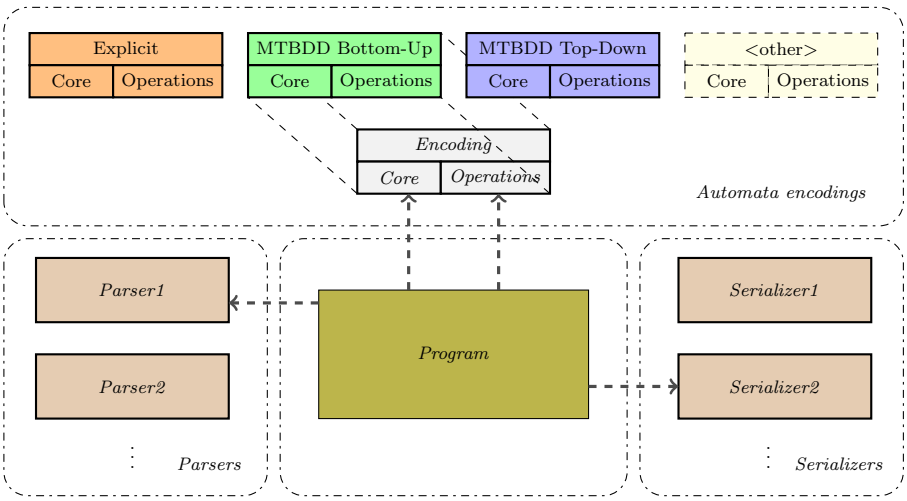


Figure 7.1: The architecture of the VATA library

At the present time, the main application of the structures and algorithms implemented in VATA for handling explicitly encoded TA is the Forester tool for verification of programs with complex dynamic data structures which is described in Chapter 4. The semi-symbolic encoding of TA has so far been used mainly for experiments with various newly proposed algorithms for handling TA.

This chapter does not present all exact details of the algorithms implemented in the library as they can be found in the referenced literature. Rather, we give an overview of the algorithms available, and most importantly, we concentrate on the various interesting optimisations that we used when implementing them. Based on experimental evidence, we argue that these optimisations are crucial for the performance of the library.

Plan of the Chapter. In Section 7.1, we present an overview of the library design. Section 7.2 introduces supported operations and also describes further optimisations, especially within the algorithms for checking language inclusion and the algorithm for computing simulation over LTSs/TA. In Section 7.3, we provide experimental evaluation of inclusion checking algorithms for both the explicit and the semi-symbolic encodings. Section 7.4 summarises the chapter and draws possible future directions.

7.1 Design of the Library

The library is designed in a modular way (see Fig. 7.1). The user can choose a module encapsulating the preferred automata encoding and its corresponding operations. Various encodings share the same general interface so it is easy to swap one encoding for another, unless encoding-specific functions or operations are taken advantage of.

Thanks to the modular design of the library, it is easy to provide an own encoding of tree (or word) automata and effectively exploit the remaining parts

of the infrastructure, such as parsers and serializers from/to different formats, the unit testing framework, performance tests, etc.

The VATA library is implemented in C++ using the Boost C++ libraries. In order to avoid expensive look-ups of entry points of virtual methods in the *virtual-method table* of an object and to fully exploit compiler's capabilities of code inlining and optimisation of code according to static analysis, the library heavily exploits polymorphism using C++ function templates instead of using virtual methods for core functions. We are convinced that this is the main reason why the performance of the optimised code (the `-O3` flag of `gcc`) is up to 10 times better than the performance of the non-optimised code (the `-O0` flag of `gcc`).

7.1.1 Explicit Encoding

In the explicit representation of TA used in VATA, top-down transitions having the form $q \xrightarrow{a} (q_1, \ldots, q_n)$ are stored in a *hierarchical data structure similar to a hash table*. More precisely, the top-level lookup table maps states to *transition clusters*. Each such cluster is itself a lookup table that maps alphabet symbols to a set of pointers to tuples of states. The set of pointers to tuples of states is represented using a red-black tree. The tuples of states are stored in a designated hash table to further reduce the required amount of space (by not storing the same tuples of states multiple times). An example of the encoding is depicted in Fig. 7.2.

Hence, in order to insert the transition $q \xrightarrow{a} (q_1, \ldots, q_n)$ into the transition table, one proceeds using the following algorithm:

1. Find a transition cluster which corresponds to the state q in the top-level lookup table. If such a cluster does not exist, create one.

2. In the given cluster, find a set of pointers to tuples of states reachable from q over a. If the set does not exist, create one.

3. Obtain the pointer to the tuple $(q_1, \ldots, q_n)$ from the tuple lookup table and insert it into the set of pointers.

If one ignores the worst-case time complexity of the underlying data structures (which, according to our experience, has usually a negligible real impact only), then inserting a single transition into the transition table requires a constant number of steps only. Yet the representation provides a more efficient encoding than a plain list of transitions because some transitions share the space required to store the parent states (e.g., state q in the transition $q \xrightarrow{a} (q_1, \ldots, q_n)$). Moreover, some transitions also share the alphabet symbol and each tuple of states appearing in the set of transitions is stored only once. Additionally, the encoding allows us to easily perform certain critical operations, such as finding a set of transitions $q \xrightarrow{a} (q_1, \ldots, q_n)$ for a given state q. This is useful, e.g., during the elimination of (top-down) unreachable states or during the top-down inclusion checking.

In some situations, one needs to manipulate many tree automata at the same time. As an example, we can mention our method for verifying programs

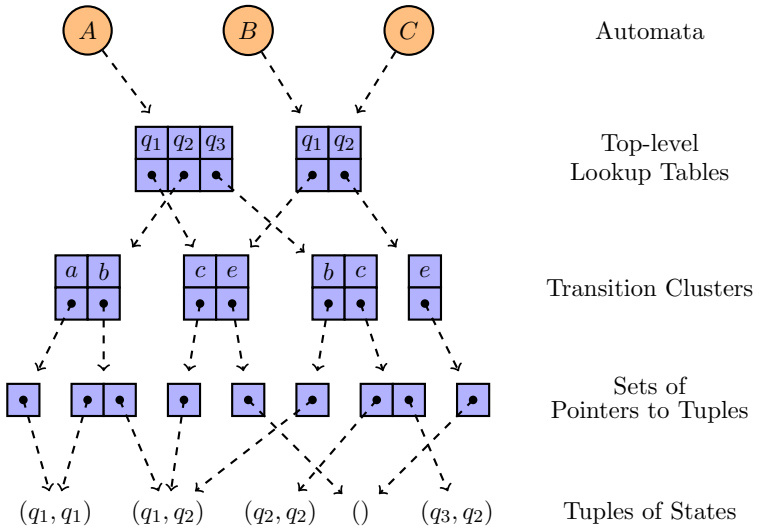


Figure 7.2: An example of the VATA's explicit encoding of transition functions of three automata A, B, C. In particular, one can see that A contains a transition $q_1 \xrightarrow{c} (q_1, q_2)$: it suffices to follow the corresponding arrows. Moreover, B also contains the same transition (and the corresponding part of the transition table is shared with A). Finally, C has the same transitions as B.

with dynamic linked data structures introduced in Chapter4 where (in theory) one needs to store one automaton representing a content of the heap for each reachable state of the program. To improve the performance of our library in such scenarios, we adapt the *copy-on-write* principle. Every time one needs to create a copy of an automaton A to be subsequently modified, it is enough to create a new automaton A' which obtains a pointer to the transition table of A (which requires constant time). Subsequently, as more transitions are inserted into A' (or A), only the part of the shared transition table which gets modified is copied (Fig. 7.2 provides an illustration of this feature).

7.1.2 Semi-Symbolic Encoding

The semi-symbolic encoding within our library uses *multi-terminal binary decision diagrams* (MTBDDs) to encode transition functions of tree automata. MTBDDs are an extension of *binary decision diagrams* (BDDs), a popular data structure for compact encoding and manipulation with Boolean formulae. In contrast to BDDs that are used to represent a function $b : \mathbb{B}^n \to \mathbb{B}$ for some $n \in \mathbb{N}$ and $\mathbb{B} = \{0, 1\}$, MTBDDs extend the co-domain to an arbitrary set S, i.e., they represent a function $m : \mathbb{B}^n \to S$.

We support two representations of semi-symbolic automata: top-down and bottom-up. The *top-down* representation (see Fig. 7.3a) maintains for each state q of a tree automaton an MTBDD that maps the binary representation of each symbol f concatenated with the binary representation of its arity n onto a set of tuples of states $T = \{(q_1, \ldots, q_n), \ldots\}$ such that for all $(q_1, \ldots, q_n) \in T$ there exist the transition $q \xrightarrow{f} (q_1, \ldots, q_n)$ in the automaton. The arity is

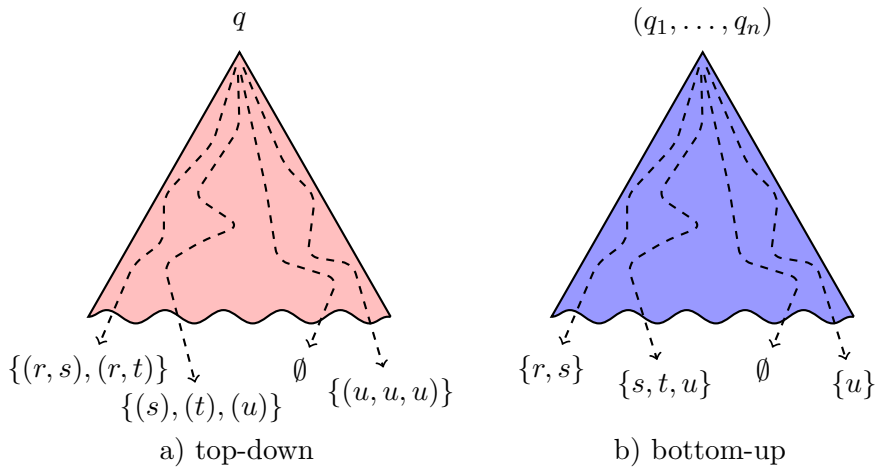


Figure 7.3: The (a) top-down and (b) bottom-up semi-symbolic encodings of transition functions. Paths in the MTBDD correspond to symbols.

encoded in the MTBDD as a part of the symbol in order to be able to distinguish between several instances of the same symbol with different arity. The library thus supports a slight extension of tree automata in which a symbol does not have a fixed arity.

The *bottom-up* representation (see Fig. 7.3b), on the other hand, maintains for each tuple $(q_1,\ldots,q_n) \in Q^*$ an MTBDD that maps the binary representation of each symbol f onto a set of states $S = \{q,\ldots\}$ such that, for all $q \in S$, it holds that the transition $(q_1,\ldots,q_n) \xrightarrow{f} q$ is in the automaton. Note that the bottom-up representation does not need to encode the arity of the symbol f into the MTBDD as it is given by the arity of the tuple for which the MTBDD is maintained. It is easy to see that the two presented encodings are mutually convertible (see [HLŠV11b] for the algorithm).

Our previous implementation of semi-symbolically represented tree automata (benchmarked in Chapter 6) used a customisation of the CUDD [Som11] library for manipulating MTBDDs. The experiments in Chapter 6 and profiling of the code showed that the overhead of the customised library is too large. Moreover, the customisation of CUDD did not provide an easy and transparent way of manipulating MTBDDs. These two facts showed that VATA would greatly benefit from a major redesign of the MTBDD back-end. Therefore, we created our own generic implementation of MTBDDs with a clean and simple-to-use interface which, according to our experiments, greatly improved the performance of many TA operations. Some of these gains were, for instance, achieved by introducing a special version of *Apply* which does not build a new MTBDD but it has a side-effect only (thus its use reduces an overhead when the result is not needed). The more elaborate description of this MTBDD package is again beyond the scope of this work, and the interested reader can find the details in [LŠV12].

7.2 Supported Operations

As we have described in the previous section, the VATA library allows a user to choose one of three available encodings: the explicit top-down, the semi-symbolic top-down, and the semi-symbolic bottom-up. Depending on the choice of the encoding, certain TA operations may or may not be available. The following operations are supported by at least one of the representations: union, intersection, elimination of (bottom-up, top-down) unreachable states, inclusion checking (bottom-up, top-down), computation of (maximum) simulation relations (downward, upward), and language preserving size reduction based on simulation equivalence. In some cases, multiple implementations of an operation are available, which is especially the case for language inclusion. This is because the different implementations are based on different heuristics that may work better for different applications as witnessed also by our experiments described in Section 7.3.

Below, we do not discuss the relatively straightforward implementation of the most basic operations on TA and we comment on the more advanced operations only.

7.2.1 Downward and Upward Simulation

Downward simulation relations can be computed over two tree automata representations in VATA: the explicit top-down and the semi-symbolic top-down encoding. The explicit variant first translates a tree automaton into a labelled transition system (LTS) as described in [ABH+08]. Then the simulation relation for this system is computed using an implementation of the state-of-the-art algorithms for computing simulations on LTSs presented in [RT07] and also in Chapter 5 with some further optimisations mentioned in Section 7.2.5. Finally, the result is projected back to the set of states of the original automaton.

The semi-symbolic variant uses a simpler simulation algorithm based on a generalisation of [INY04] to trees.

Upward simulation can currently be computed over the explicit representation only. The computation is again performed via a translation to an LTS (the details are in [ABH+08]), and the relation is computed using the engine for computing simulation relations on LTSs as above.

7.2.2 Simulation-based Size Reduction

In a typical setting, one often wants to use a representation of tree automata that is as small as possible in order to reduce the memory consumption and/or speed up operations on the automata (especially the potentially costly ones, such as inclusion testing). To achieve that, the classical approach is to use determinisation and minimisation. However, the minimal deterministic tree automata can still be much bigger than the original nondeterministic ones. Therefore, VATA offers a possibility to reduce the size of tree automata without determinisation by their quotienting w.r.t. an equivalence relation—currently,

the library uses downward simulation simulation equivalence, but the user can easily provide an arbitrary relation suitable for collapsing states.

The procedure works as follows: first, the downward simulation relation $\preceq_D$ is computed for the automaton. Then, the symmetric fragment of $\preceq_D$ (which is an equivalence) is extracted, and each state appearing within the transition function is replaced by a representative of the corresponding equivalence class. A further reduction is then based on the following observation: if an automaton contains a transition $q \xrightarrow{a} (q_1, \ldots, q_n)$, any additional transition $q \xrightarrow{a} (r_1, \ldots, r_n)$ where $r_i \preceq_D q_i$ can be omitted since it does not contribute to the language of the result (recall that, for the downward simulation preorder $\preceq_D$, it holds that $q \preceq_D r \implies \mathcal{L}(q) \subseteq \mathcal{L}(r)$).

7.2.3 Bottom-up Inclusion

Bottom-up inclusion testing is implemented for the explicit top-down and the semi-symbolic bottom-up representation in VATA. As its name suggests, the algorithm naturally proceeds in the bottom-up way, therefore the top-down encoding is not very suitable here. In the case of the explicit representation, however, one can afford to build a temporary bottom-up encoding since the overhead of such a translation is negligible compared to the complexity of following operations.

Both the explicit and semi-symbolic version of the bottom-up inclusion algorithm are based on the approach introduced in [BHH$^+$08]. Here, the main principle used for checking whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ is to search for a tree which is accepted by $\mathcal{A}$ and not by $\mathcal{B}$ (thus being a witness for $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$). This is done by simultaneously traversing both $\mathcal{A}$ and $\mathcal{B}$ from their leaf rules while generating pairs $(p_\mathcal{A}, P_\mathcal{B}) \in Q_\mathcal{A} \times 2^{Q_\mathcal{B}}$ where $p_\mathcal{A}$ represents a state into which $\mathcal{A}$ can get on some input tree and $P_\mathcal{B}$ is the set of *all* states into which $\mathcal{B}$ can get over the same tree. The inclusion then does clearly not hold iff it is possible to generate a pair consisting of an accepting state of $\mathcal{A}$ and of exclusively non-accepting states of $\mathcal{B}$.

The algorithm collects the so far generated pairs $(p_\mathcal{A}, P_\mathcal{B})$ in a set called $Visited$. Another set called $Next$ is used to store the generated pairs whose successors are still to be explored. One can then observe that whenever one can reach a counterexample to inclusion from $(p_\mathcal{A}, P_\mathcal{B})$, one can also reach a counterexample from any $(p_\mathcal{A}, P'_\mathcal{B} \subseteq P_\mathcal{B})$ as $P'_\mathcal{B}$ allows less runs than $P_\mathcal{B}$. Using this observation, both mentioned sets can be represented using antichains. In particular, one does not need to store and further explore any two elements comparable w.r.t. $(=, \subseteq)$, i.e., by equality on the first component and inclusion on the other component.

In the following we describe several modifications of the existing bottom-up inclusion algorithm (proposed in [BHH$^+$08]) which according to experiments substantialy improve the performance of inclusion queries—see Algorithm 7.

Clearly, the running time of the above algorithm strongly depends on the total number of pairs $(p_\mathcal{A}, P_\mathcal{B})$ taken from $Next$ for further processing. Indeed, this is one of the reasons why the antichain-based optimisations helps. According to our experience, the number of pairs which needs to be processed can further

Algorithm 7: Bottom-up Inclusion Algorithm

Input: TA $\mathcal{A}_S = (Q_S, \Sigma, \Delta_S, F_S)$, $\mathcal{A}_B = (Q_B, \Sigma, \Delta_B, F_B)$, $\preceq \subseteq (Q_S \cup Q_B)^2$
Output: *true* if $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_B)$, *false* otherwise

1 $Visited := \{(p_S, P_B) : a \to p_S \in \Delta_S \wedge P_B = up_a(\{()\})^{\preceq}\}$;
2 **if** $\exists (p_S, P_B) \in Visited.p_S \in F_S \wedge P_B \cap F_B = \emptyset$ **then**
3 **return** *false*;
4 $Next := Visited$;
5 **while** $Next \neq \emptyset$ **do**
6 **pick** $(p_S, P_B) \in Next$ **such that** $\forall (p'_S, P'_B) \in Next.|P_B| \leq |P'_B|$;
7 $Next := Next \setminus \{(p_S, P_B)\}$;
8 **foreach** $a(q_1, \ldots, q_n) \to p'_S \in \Delta_S$ **such that** $\exists k.q_k = p_S$ **do**
9 **foreach** $Q_1, \ldots, Q_n$ **such that** $(q_i, Q_i) \in Visited \wedge Q_k = P_B$ **do**
10 $P'_B := up_a(Q_1 \times \cdots \times Q_n)^{\preceq}$;
11 **if** $p'_S \in F_S \wedge P'_B \cap F_B = \emptyset$ **then**
12 **return** *false*;
13 **if** $\exists p'_B \in P'_B.p'_S \preceq p'_B$ **then**
14 **continue**;
15 **if** $\exists (p''_S, P''_B) \in Visited$ **such that** $p''_S \preceq p'_S \wedge P'_B \preceq^{\forall\exists} P''_B$ **then**
16 **continue**;
17 $Visited := (Visited \setminus \{(p''_S, P''_B) : p'_S \preceq p''_S \wedge P''_B \preceq^{\forall\exists} P'_B\}) \cup \{(p'_S, P'_B)\}$;
18 $Next := (Next \setminus \{(p''_S, P''_B) : p'_S \preceq p''_S \wedge P''_B \preceq^{\forall\exists} P'_B\}) \cup \{(p'_S, P'_B)\}$;
19 **return** *true*;

be reduced when processing the pairs stored in $Next$ in a suitable order. Our experimental results have shown that we can achieve a very good improvement by preferring those pairs $(p_\mathcal{A}, P_\mathcal{B})$ which have smaller (w.r.t. the size of the set) second component (line 6).

Yet another way that we found useful when improving the above algorithm is to optimise the way the algorithm computes the successors of a pair from $Next$. The original algorithm picks a pair $(p_\mathcal{A}, P_\mathcal{B})$ from $Next$ and puts it into $Visited$. Then, it finds all transitions of the form $(p_{\mathcal{A},1}, \ldots, p_{\mathcal{A},n}) \xrightarrow{a} p$ in $\mathcal{A}$ such that $(p_{\mathcal{A},i}, P_{\mathcal{B},i}) \in Visited$ for all $1 \leq i \leq n$ and $(p_{\mathcal{A},j}, P_{\mathcal{B},j}) = (p_\mathcal{A}, P_\mathcal{B})$ for some $1 \leq j \leq n$. For each such transition, it finds all transitions of the form $(q_1, \ldots, q_n) \xrightarrow{a} q$ in $\mathcal{B}$ such that $q_i \in P_{\mathcal{B},i}$ for all $1 \leq i \leq n$. Here, the process of finding the needed $\mathcal{B}$ transitions is especially costly (lines 8 and 9). In order to speed it up, we cache for each alphabet symbol a, each position i, and each set $P_{\mathcal{B},i}$, the set of transitions $\{(q_1, \ldots, q_n) \xrightarrow{a} q \in \Delta_\mathcal{B} : q_i \in P_{\mathcal{B},i}\}$ at the first time it is used in the computation of successors. Then, whenever we need to find all transitions of the form $(q_1, \ldots, q_n) \xrightarrow{a} q$ in $\mathcal{B}$ such that $q_i \in P_{\mathcal{B},i}$ for all $1 \leq i \leq n$, we find them simply by intersecting the sets of transitions cached for each $(P_{\mathcal{B},i}, i, a)$.

Next, we propose another modification of the algorithm which aims to improve the performance especially in those cases where finding a counterexample to inclusion requires us to build representatives of trees with higher depths or in the cases where the inclusion holds. Unlike the original approach which moves only one pair $(p_\mathcal{A}, P_\mathcal{B})$ from $Next$ to $Visited$ at the beginning of each iteration of the main loop, we add the newly created pairs $(p_\mathcal{A}, P_\mathcal{B})$ into $Next$

and $Visited$ at the same time (immediately after they are generated). This, according to our experiments, allows $Visited$ to converge faster towards the fixpoint (lines 17 and 18).

Finally, another optimisation of the algorithm presented in [BHH+08] appeared in [ACH+10]. This optimisation maintains the sets $Visited$ and $Next$ as antichains w.r.t. $(\preceq_U, \succeq_U^{\exists\forall})$[2]. Hence, more pairs can be discarded from these sets. Moreover, for pairs that cannot be discarded, one can at least reduce the sets on their right-hand side by removing states that are simulated by some other state in these sets (this is based on the observation that any tree accepted from an upward-simulation-smaller state is accepted from an upward-simulation-bigger state too). Finally, one can also use upward simulations between states of the two automata being compared. Then, one can discard any pair $(p_{\mathcal{A}}, P_{\mathcal{B}})$ such that there is some $p_{\mathcal{B}} \in P_{\mathcal{B}}$ that upward-simulates p_A because it is then clear that no tree can be accepted from $p_{\mathcal{A}}$ that could not be accepted from $p_{\mathcal{B}}$. All these optimisations are also available in VATA and can optionally be used—they are not used by default since the computation of the upward simulation can be quite costly.

7.2.4 Top-down Inclusion

Top-down inclusion checking is supported by the explicit top-down and semi-symbolic top-down representations in VATA. As we have already discussed in Chapter 6, when one tries to solve inclusion of TA languages top-down in a naïve way, using a plain subset-construction-like approach, one immediately hits a problem due to the top-down successors of particular states are *tuples* of states. Hence, after one step of the construction, one needs to check inclusion on tuples of states, then tuples of tuples of states, etc. However, there is a way how to get out of this trap as shown in [HVP05] and also in Chapter 6. To recall from Chapter 6, the main idea of the approach resembles a conversion from the *disjunctive normal form* (DNF) to the *conjunctive normal form* (CNF) taking into account that top-down transitions of tree automata form a kind of and-or graphs (the disjunctions are between top-down transitions and conjunctions among the successors within particular transitions).

VATA contains an implementation of the top-down inclusion checking algorithm as presented in Chapter 6. This algorithm uses several optimisations, e.g., caching of results of auxiliary language inclusion queries between states of the automata whose languages are being compared. More precisely, when checking whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ holds for two tree automata $\mathcal{A}$ and $\mathcal{B}$, the algorithm stores a set of pairs $(p_{\mathcal{A}}, P_{\mathcal{B}}) \in Q_{\mathcal{A}} \times 2^{Q_{\mathcal{B}}}$ for which the language inclusion $\mathcal{L}(p_{\mathcal{A}}) \subseteq \mathcal{L}(P_{\mathcal{B}})$ has been shown *not* to hold. As a further optimisation, the set is stored as an antichain based on comparing the states w.r.t. the downward simulation preorder. The use of the downward simulation is one of the main advantages of this approach compared with the bottom-up inclusion checking since this preorder is cheaper to compute and usually richer than the

[2] One says that $P \preceq_U^{\exists\forall} Q$ holds iff $\forall p \in P\ \exists q \in Q :\ p \preceq_U q$. Note also that the upward simulation must be parameterised by the identity in this case [ACH+10].

upward simulation. Indeed, as we have shown in Chapter 6, the top-down inclusion checking is often—though not always—superior to bottom-up inclusion checking.

Moreover, VATA also imlements an optimized version of the top-down inclusion checking algorithm that extends the previous version by caching even the pairs $(p_{\mathcal{A}}, P_{\mathcal{B}}) \in Q_{\mathcal{A}} \times 2^{Q_{\mathcal{B}}}$ for which the language inclusion $\mathcal{L}(p_{\mathcal{A}}) \subseteq \mathcal{L}(P_{\mathcal{B}})$ has been shown to hold. This extension is far from trivial since the caching must be done very carefully in order to avoid a sort of circular reasoning when answering the various auxiliary language inclusion queries. A precise description of this rather involved algorithm is again described in Chapter 6. As our experiments show, the caching positive pairs comes with some overhead, which does not allow it to always win over the previous algorithm, but there are still many cases in which it performs significantly better.

7.2.5 Computing Simulation over LTS

The explicit part of VATA uses a highly optimised LTS simulation algorithm proposed in [RT07] together with the improvements presented in 5. The main idea of the algorithm is to start with an overapproximation of the simulation preorder (a possible initial approximation is the relation $Q \times Q$) which is then iteratively pruned whenever it is discovered that the simulation relation cannot hold for certain pairs of states. For a better efficiency, the algorithm represents the current approximation R of the simulation being computed using a so-called *partition-relation pair*. The partition splits the set of states into subsets (called *blocks*) whose elements are equivalent w.r.t. R, and a relation obtained by lifting R to blocks.

In order to be able to deal with the partition-relation pair efficiently, the algorithm needs to record for each block a matrix of counters of size $|Q||\Sigma|$ where, for the given LTS, Q is the set of states and Σ is the set of labels. The counters are used to count how many transitions going from the given state via a given symbol a lead to states in the given block (or blocks currently considered to be bigger w.r.t. the simulation). This information is then used to optimise re-computation of the partition-relation pair when pruning the current approximation of the simulation relation being computed (for details see, e.g., [RT07]). Since the number of blocks can (and often does) reach the number of states, the naïve solution requires $|Q|^2|\Sigma|$ counters in the worst case. It turns out that this is one of the main barriers which prevents the algorithm from scaling to systems with large alphabets and/or large sets of states.

Working towards a remedy for the above problem, one can observe that the mentioned algorithm actually works in several phases. At the beginning, it creates an initial estimation of the partition-relation pair which typically contains large equivalence classes. Then it initialises the counters for each element of the partition. Finally, it starts the iterative partition splitting. During this last phase, the counters are only decremented or copied to the newly created blocks. Moreover, the splitting of some block is itself triggered by decrementing some set of counters to 0. In practice, late phases of the iteration often witness

a lot of small equivalence classes having very sparsely populated counters with 0 being the most abundant value.

This suggests that one could use sparse matrices containing only the non-zero elements. Unfortunately, according to our experience, this turns out to be the worst possible solution which strongly degrades the performance. The reason is that the algorithm accesses the counters very frequently (the values of the counters are initialised at the beginning and then the algorithm keeps decrementing them by one), hence any data structure with non-constant time access causes the computation to stall. A somewhat better solution is to record the non-zero counters using a hash table, but the memory requirements of such representation are not yet reasonable.

Instead, we are currently experimenting with storing the counters in blocks, using a copy-on-write approach and a zeroed-block deallocation. In short, we divide the matrix of counters into a list of blocks of some fixed size. Each block contains an additional counter (a block-level counter) which sums up all the elements within the block. As soon as a block contains a single non-zero counter only, it can safely be deallocated—the content of the non-zero counter is then recorded in the block-level counter.

Our initial experiments show that, using the above approach, one can easily reduce the memory consumption by the factor of 5 for very large instances of the problem compared to the array-based representation which was used for experiments presented in Chapter 5. The best value to be used as the size of blocks of counters is still to be studied—after some initial experiments, we are currently using blocks of size $\sqrt{|Q|}$.

7.3 Experimental Evaluation

In order to illustrate the level of optimisation that has been achieved in VATA and that can be exploited in its applications (like the Forester tool [HHR+11a]), we compared its performance against Timbuk and the prototype library considered in Chapter 6, which—despite its prototype status—already contained a quite efficient TA implementation.

The comparison of performance of VATA (using the explicit encoding) and Timbuk was done for union and intersection of more than 3,000 pairs of TA. On average, VATA was over 20,000 times faster on union and over 100,000 times faster on intersection.

When comparing VATA with the implementation used in Chapter 6, we concentrated on language inclusion testing which is one of the most costly operations on nondeterministic TA. In particular, we conducted a set of experiments evaluating the performance of the VATA's optimised TA language inclusion algorithms on pairs of TA obtained from *abstract regular tree model checking* of the algorithm for rebalancing red-black trees after insertion or deletion of a leaf node (which is the same test set that was used in Chapter 6).

Table 7.1: Experiments with inclusion for the explicit encoding

	inclusion					
	mixed		does not hold		holds	
	winner	timeouts	winner	timeouts	winner	timeouts
`explup`	24.14 %	0.00 %	24.84 %	0.00 %	16.85 %	0.00 %
`explup+s`	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
`expldown`	36.35 %	32.51 %	39.85 %	26.01 %	0.00 %	90.80 %
`expldown+s`	4.15 %	18.27 %	0.00 %	20.31 %	47.28 %	0.00 %
`expldown-opt`	32.20 %	32.51 %	35.30 %	26.01 %	0.00 %	90.80 %
`expldown-opt+s`	3.15 %	18.27 %	0.00 %	20.31 %	35.87 %	0.00 %

7.3.1 Explicit Encoding

For the explicit encoding, we measured for each inclusion method the fraction of cases in which the method was the fastest among the evaluated methods on the set of almost 2000 tree automata pairs. The results of this experiment are given in Table 7.1. The rows are labelled as follows: row `expldown` is for pure downward inclusion checking, row `expldown+s` is for downward inclusion using downward simulation, `expldown-opt` is a row for pure downward inclusion checking with the optimisation proposed in Section 7.2.4, and row `expldown-opt+s` is downward inclusion checking with simulation using the same optimisation. Rows `explup` and `explup+s` give the results for pure upward inclusion checking and upward inclusion checking with simulation respectively. The timeout was set to 30 s.

We also checked the performance of the algorithms for cases when inclusion either *does* or *does not* hold in order to explore the ability of the algorithms to either find a counterexample in the case when inclusion does not hold, or prove the inclusion in case it does. These results are given in separate columns of Table 7.1.

When compared to our previous implementation benchmarked in the Chapter 6, VATA performed almost always better. The average speed-up was even as high as 200 times for pure downward inclusion checking. The old implementation was faster in about 2.5 % of the cases, and the difference was not significant. We can observe that unlike the results presented in Chapter 6 the computation of `explup` no longer exceeds the available time thanks to our improvements. The table is still dominated by downward inclusion which however does not finish the computation in some cases. The combination of downward inclusion with downward simulation allows to reduce the number of timeouts especially when one considers the cases in which the inclusion holds.

7.3.2 Semi-Symbolic Encoding

We have performed a set of similar experiments for the semi-symbolic encoding, the results of which are given in Table 7.2. The rows are labelled as follows: row `symdown` is for pure downward inclusion checking, row `symdown+s` is for downward inclusion using downward simulation, `symdown-opt` is a row for pure downward inclusion checking with the optimisation proposed in Section 7.2.4

Table 7.2: Experiments with inclusion for the semi-symbolic encoding

	inclusion					
	mixed		does not hold		holds	
	winner	timeouts	winner	timeouts	winner	timeouts
`symup`	24.25 %	22.26 %	21.91 %	23.39 %	80.26 %	0.00 %
`symdown`	44.02 %	5.87 %	45.03 %	2.48 %	19.74 %	72.37 %
`symdown+s`	0.00 %	77.93 %	0.00 %	80.03 %	0.00 %	36.84 %
`symdown-opt`	31.73 %	5.87 %	33.06 %	2.48 %	0.00 %	72.37 %
`symdown-opt+s`	0.00 %	78.00 %	0.00 %	80.09 %	0.00 %	36.84 %

and row `symdown-opt+s` is downward inclusion checking with simulation using the same optimisation. Row `symup` gives the results for pure upward inclusion checking. The column "winner" shows the percentage of cases in which the corresponding method was the fastest. The column "timeout" shows how often the method failed to compute the result within the given timeout which was again set to 30 s.

As in the experiments for the explicit encoding, we also checked the performance of the algorithms for cases when inclusion either *does* or *does not* hold. These results are given in the separate columns of Table 7.2.

When compared to our previous implementation benchmarked in Chapter 6, VATA again performs significantly better, with the pure upward inclusion being on average over 300 times faster and the pure downward inclusion being even over 3000 times faster.

7.4 Conclusions and Future Work

In this chapter, we have introduced and described a new efficient and open-source nondeterministic tree automata library that supports both explicit and semi-symbolic encoding of the tree automata transition function. The semi-symbolic encoding makes use of our own MTBDD package instead of the previously used customisation of the CUDD library.

In the future, it would be interesting to try to implement a simulation-aware symbolic encoding of antichains using BDDs. Further, an implementation of other TA operations, such as determinisation (which, however, is generally desired to be avoided), or complementation (without a need of determinisation) could also be important for certain applications such as the decision procedures of WSkS or MSO.

Finally, we hope that a public release of our library will attract more people to use it and even better contribute to the code base. Indeed, we believe that the library is written in a clean and understandable way that should make such contributions possible.

8 Conclusions and Future Directions

Detailed conclusions of each specific topic discussed in the thesis have been given at the end of the corresponding chapters. Here, we summarise once more the main points and discuss possible future directions of the research from a broader perspective.

8.1 A Summary of the Contributions

This thesis focuses on formal verification of programs manipulating complex dynamic data structures. Inspired by the existing techniques based on using tree automata ([BHRV06b]) and also techniques based on separation logic ([Rey02]), we have developed a novel approach which tries to combine ideas from these lines of research.

In particular, we have proposed an encoding of sets of heaps using an original notion of forest automata. Essentially, a heap is split into a tuple of trees such that non-tree links can be represented via explicit references to the roots of the created trees. A forest automaton is then basically a tuple of ordinary tree automata representing a set of heaps decomposed in this way. To obtain some concrete heap out of this representation, one can pick a tuple of trees from the languages of the tree automata which an FA consists of and replace the non-local references by gluing the corresponding nodes.

Plain forest automata can represent sets of heaps which can be decomposed into a finite number of tree components. In order to extend the expressive power of the formalism, the original concept has been further extended by allowing hierarchically nested forest automata to appear within the alphabet. We have shown that hierarchical forest automata can indeed represent some sets of heaps which would normally require an unbounded number of tree components and that are important in practice (e.g., DLLs).

As the next step, we have developed a new symbolic verification method in which we finitely represent infinite sets of heap configurations using forest automata. During a verification run, program statements are interpreted directly over forest automata such that we compute the effect of a particular program statement on infinitely many configurations in one step.

Our verification technique is based on the use of non-deterministic finite tree automata, and the performance of the approach strongly depends on the efficiency of the tree automata operations used. Among them, size reduction and language inclusion are especially critical since they used to be traditionally implemented via determinisation and only recently started to be implemented directly on NTA (to avoid the exponential cost of determinisation as much as possible). Therefore, we have also invested into improving the state-of-the-art algorithms for these operations.

The size reduction algorithm that we use is based upon collapsing states of an automaton according to a suitable preorder. We, in particular, use downward simulation which can be computed via a translation of a TA into a labelled transition system. The original algorithm for computing simulation over LTSs presented in [ABH+08, AHKV08] is a straightforward extension of the algorithm for Kripke structures presented in [HHK95, RT07]. We show that the increase in its complexity caused by the introduction of transition labels can be to a large degree eliminated, which is supported by our experimental results.

Furthermore, we have also intensively investigated methods for checking language inclusion of tree automata which is indirectly used for checking inclusion of forest automata. The approach of [ACH+10] shows how one can combine simulations and antichains for checking language inclusion of finite word and also tree automata. This allows to achieve great computation speedups, especially when finite word automata are considered. In the case of finite tree automata, bottom-up inclusion is considered in [ACH+10]. This can only be combined with upward simulation which is quite expensive to compute and usually does not yield bigger gains in speed. In order to solve this problem, we have generalised the top-down approach of [TH03] to tree automata of arbitrary arity. Moreover, we have extended the algorithm by using antichains combined with downward simulation which is cheaper to compute and allows for a better speedup.

Finally, we have created a freely available tree automata library containing an efficient implementation of many important general purpose algorithms for explicitly and semi-symbolically represented tree automata. For this purpose, the basic versions of the data structures and algorithms described in the above works have been carefully optimised, which we have also described in Chapter 7.

8.2 Further Directions

There are numerous directions of further work in the areas covered by the thesis. From the theoretical perspective, the expressive power of hierarchical forest automata is an interesting question which has not yet been systematically discussed. In connection to that, there arises a question of allowing recursively nested FA which would extend the expressive power to other interesting sets of graphs. It is, however, not yet clear how such an extension should look like such that it allows for the needed automata operations to be implemented over it. Further, the proposed algorithm for backward symbolic execution should be implemented and experimentally evaluated within the framework of predicate language abstraction. One problem that could arise here is that of the precision of the intersection under-approximation. If it appears problematic in some practical cases, some more precise solution will have to be sought. Likewise, if the precision of the currently used algorithm for inclusion checking of FA appears insufficient on some examples (which has not yet happened), it may turn out useful to increase its precision by taking into account the hierarchical structuring of FA. A related theoretical question is whether or not the inclusion is decidable (even if neglecting the cost of such a check).

A very broad area for further research is that of extending the proposed techniques to be able to cope with data stored inside the dynamically linked data structures (which is crucial for verification of programs over red-black trees, unmodified skip lists, or various user-specific scenarios exploiting dynamically linked data structures) as well as for dealing with concurrency and/or recursion (without the restrictions imposed in Chapter 4).

Next, concerning the problem of computing simulations for LTSs, it would be interesting to study whether the memory requirements of the algorithm can be further reduced by using some sophisticated data structure—such as BDDs—for storing internal data of the algorithm. Such an approach could perhaps reduce the impact of the alphabet size of the LTS even more. Moreover, an interesting subject for further work is to go beyond reduction of automata based on collapsing simulation equivalent states. Indeed, sometimes, it is useful to split some states allowing a subsequent collapsing to be much more productive.

The problem of language inclusion of TA is also a possible subject of further research focus. The bottom-up approach does not seem to benefit from the combination with upward simulation. Therefore, it would be interesting to see whether there exists a different (and possibly cheaper to compute) relation which could improve the performance of upward inclusion checking. On the other hand, despite the optimisations that we proposed, the top-down approach can suffer from an explosion of the number of downward successors of a given macro state. Moreover, in many cases, not all the successors need to be examined if one is able to explore them in a suitable order. Possibilities of optimising the sequence of successors are therefore also an interesting subject of future work.

The further development of our general purpose TA library involves implementation of so far missing operations such as determinisation, complementation, general-purpose transduction, etc. Here, complementation is special in that we are not aware of any existing efficient way how to implement it without determinisation (although some initial ideas have appeared, e.g., in [Hos10]). At the same time, complementation is crucial for some automata-based algorithms such as the decision procedures of WSkS or MSO. For that reason, it would be nice to either find some efficient way how to complement automata without determinising them or to find ways how to avoid explicit complementation as much as possible. Apart from that, specialized versions of algorithms working specifically with finite word automata (which are themselves a special kind of TA) could be introduced in order to handle them more efficiently.

8.3 Publications and Tools Related to this Work

The verification technique for programs manipulating complex dynamic data structures and the underlying formalism of forest automata presented in Chapter 3 and Chapter 4 were first introduced in [HHR$^+$11a]. An extended version of the original description appeared in [HHR$^+$12]. The proposed improvements of the in algorithm for computing simulations over labelled transition systems were published in [HŠ09a]. An extended version then appeared in [HŠ10]. The

proposed top-down inclusion checking algorithm combined with antichains and downward simulation was described in [HLŠV11a]. Finally, our work on the general purpose tree automata library (and the optimised data structures and algorithms that it is based on) was presented in [LŠV12].

Apart from that, the full versions of some of the above mentioned papers were published as technical reports [HHR+11b, HŠ09b, HLŠV11b]. A detailed description of the algorithms for computing abstraction (Section 4.4) as well as the automatic discovery of nested FA (Section 4.3) have not yet been published (and are planned to be published later on). The proposed techniques were implemented in the Forester tool and the VATA library publicly available over the internet[1][2].

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`
[2] `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`

References

[ABC+08] P.A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic Abstraction for Programs with Dynamic Memory Heaps. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 341–354, Berlin, Heidelberg, 2008. Springer Verlag.

[ABH+08] P.A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. Computing Simulations over Tree Automata: Efficient Techniques for Reducing Tree Automata. In *Proc. of TACAS*, number 4963 in LNCS, pages 93–108, Berlin, Heidelberg, 2008. Springer Verlag.

[ACH+10] P.A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When Simulation Meets Antichains (on Checking Language Inclusion of NFAs). In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 158–174, Berlin, Heidelberg, 2010. Springer Verlag.

[ACV11] P.A. Abdulla, J. Cederberg, and T. Vojnar. Monotonic Abstraction for Programs with Multiply-Linked Structures. In *Proc. of RP*, volume 6945 of *LNCS*, pages 125–138, Berlin, Heidelberg, 2011. Springer Verlag.

[AHKV08] P.A. Abdulla, L. Holík, L. Kaati, and T. Vojnar. A Uniform (Bi-)Simulation-Based Framework for Reducing Tree Automata. In *Proc. of MEMICS*, pages 3–11. Faculty of Informatics MU, 2008.

[AJMd02] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, volume 2404 of *LNCS*, Berlin, Heidelberg, 2002. Springer Verlag.

[ALdR05] P. A. Abdulla, A. Legay, J. d'Orso, and A. Rezine. Simulation-Based Iteration of Tree Transducers. In *Proc. of TACAS*, volume 3440 of *LNCS*, pages 30–44, Berlin, Heidelberg, 2005. Springer Verlag.

[BBH+11] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomáš Vojnar. Programs with Lists are Counter Automata. *Formal Methods in System Design*, 38(2):158–192, April 2011.

[BCC+07] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'hearn, and H. Yang. Shape analysis for composite data structures. In

Proc. of CAV, pages 178–192, Berlin, Heidelberg, 2007. Springer Verlag.

[BHH+08] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In *Proc. of CIAA'08*, volume 5148 of *LNCS*, Berlin, Heidelberg, 2008. Springer Verlag.

[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*, Berlin, Heidelberg, 2005. Springer Verlag.

[BHRV06a] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. A preliminary version was presented at Infinity'05.

[BHRV06b] 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*, pages 52–70, Berlin, Heidelberg, 2006. Springer Verlag.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV*, volume 3114 of *LNCS*, pages 372–386, Berlin, Heidelberg, 2004. Springer Verlag.

[Bou11] T. Bourdier. Tree Automata-based Semantics of Firewalls. In *Proc. of SAR-SSI*, pages 171–178, Washington, DC, USA, 2011. IEEE Computer Society.

[Bry86] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computing*, 35(8):677–691, 1986.

[BT02] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV*, volume 2404 of *LNCS*, pages 539–554, Berlin, Heidelberg, 2002. Springer Verlag.

[CDOY09] C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-abduction. In *Proc. of PLDI*, volume 44 of *ACM SIGPLAN Notices*, pages 289–300, New York, NY, USA, 2009. ACM Press.

[CJH+09] P. Claves, D. Jansen, S.J. Holtrup, M. Mohr, A. Reis, M. Schatz, and I. Thesing. The LETHAL Library. `http://lethal.sourceforge.net/`, 2009.

[CMZ+97] E. M. Clarke, K. L. Mcmillan, X. Zhao, M. Fujita, and J. Yang. Spectral Transforms for Large Boolean Functions withApplications to Technology Mapping. *Formal Methods in System Design*, 10(2-3):137–148, April 1997.

[CRN07] B.-Y. Chang, X. Rival, and G. Necula. Shape Analysis with Structural Invariant Checkers. In *Proc. of SAS*, volume 4634 of *LNCS*, pages 384–401, Berlin, Heidelberg, 2007. Springer Verlag.

[DEG06] J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS*, volume 3920 of *LNCS*, pages 27–41, Berlin, Heidelberg, 2006. Springer Verlag.

[DGG93] D. Dams, O. Grumberg, and R. Gerth. Generation of Reduced Models for Checking Fragments of CTL. In *Proc. of CAV*, volume 697 of *LNCS*, pages 479–490, Berlin, Heidelberg, 1993. Springer Verlag.

[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*, volume 6806 of *LNCS*, pages 372–378, Berlin, Heidelberg, 2011. Springer Verlag.

[DR10] L. Doyen and J.-F. Raskin. Antichain Algorithms for Finite Automata. In *Proc. of TACAS*, volume 6015 of *LNCS*, pages 2–22, Berlin, Heidelberg, 2010. Springer Verlag.

[DWDHR06] M. De Wulf, L. Doyen, T. Henzinger, and J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proc. of CAV*, volume 4144 of *LNCS*, pages 17–30, Berlin, Heidelberg, 2006. Springer Verlag.

[Gen03] T. Genet. Timbuk/Taml: A Tree Automata Library. `http://www.irisa.fr/lande/genet/timbuk`, 2003.

[GL94] O. Grumberg and D.E. Long. Model Checking and Modular Verification. *TOPLAS*, 16(3):843–871, May 1994.

[GVA07] B. Guo, N. Vachharajani, and D.I. August. Shape Analysis with Inductive Recursion Synthesis. In *Proc. of PLDI*, volume 42 of *ACM SIGPLAN Notices*, pages 256–265, New York, NY, USA, 2007. ACM Press.

[HHK95] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *Proc. of FOCS*, pages 453–462, Washington, DC, USA, 1995. IEEE Computer Society.

[HHR+11a] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV*, volume 6806 of *LNCS*, pages 424–440, Berlin, Heidelberg, 2011. Springer Verlag.

[HHR+11b] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. Technical report, Faculty of Information Technology, BUT, 2011.

[HHR+12] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest automata for verification of heap manipulation. *Formal Methods in System Design*, pages 1–24, 2012.

[HLŠV11a] L. Holík, O. Lengál, J. Šimáček, and T. Vojnar. Efficient Inclusion Checking on Explicit and Semi-symbolic Tree Automata. In *Proc. of ATVA*, volume 6996 of *LNCS*, pages 243–258, Berlin, Heidelberg, 2011. Springer Verlag.

[HLŠV11b] L. Holík, O. Lengál, J. Šimáček, and T. Vojnar. Efficient Inclusion Checking on Explicit and Semi-Symbolic Tree Automata. Technical report, Faculty of Information Technology, BUT, 2011.

[Hol11] L. Holík. *Simulations and Antichains for Efficient Handling of Finite Automata*. PhD thesis, Faculty of Information Technology, Brno University of Technology, 2011.

[Hos10] H. Hosoya. *Foundations of XML Processing: The Tree-Automata Approach*. Cambridge University Press, 2010.

[HŠ09a] L. Holík and J. Šimáček. Optimizing an LTS-Simulation Algorithm. In *Proc. of MEMICS*, pages 93–101. Faculty of Informatics MU, 2009.

[HŠ09b] L. Holík and J. Šimáček. Optimizing an LTS-Simulation Algorithm. Technical Report FIT-TR-2009-03, Brno University of Technology, 2009.

[HŠ10] L. Holík and J. Šimáček. Optimizing an LTS-Simulation Algorithm. *Computing and Informatics*, 2010(7):1337–1348, 2010.

[HVP05] H. Hosoya, J. Vouillon, and B.C. Pierce. Regular Expression Types for XML. *TOPLAS*, 27(1):46–90, January 2005.

[INY04] L. Ilie, G. Navarro, and S. Yu. On NFA Reductions. In *Theory Is Forever*, volume 3113 of *Lecture Notes in Computer Science*, pages 112–124, Berlin, Heidelberg, 2004. Springer Verlag.

[KMS01] N. Klarlund, A. Møller, and M. Schwartzbach. MONA Implementation Secrets. In *Proc. of CIAA*, volume 2088 of *LNCS*, pages 182–194, Berlin, Heidelberg, 2001. Springer Verlag.

[LŠV12] O. Lengál, J. Šimáček, and T. Vojnar. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Proc. of TACAS*, volume 7214 of *LNCS*, pages 79–94, Berlin, Heidelberg, 2012. Springer Verlag.

[MPQ11] P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. In *Proc. of POPL*, volume 46 of *ACM SIGPLAN Notices*, pages 611–622, New York, NY, USA, 2011. ACM Press.

[MS01] A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI*, volume 36 of *ACM SIGPLAN Notices*, pages 221–231, New York, NY, USA, 2001. ACM Press.

[MTLT10] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic Numeric Abstractions for Heap-manipulating Programs. In *Proc. of POPL*, volume 45 of *ACM SIGPLAN Notices*, pages 211–222, New York, NY, USA, 2010. ACM Press.

[NDQC07] H.H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *Proc. of VMCAI*, volume 4349 of *LNCS*, pages 251–266, Berlin, Heidelberg, 2007. Springer Verlag.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[PT87] R. Paige and R.E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[Pug90] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[Rey02] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[RT07] F. Ranzato and F. Tapparo. A New Efficient Simulation Equivalence Algorithm. In *Proc. of LICS*, pages 171–180, Washington, DC, USA, 2007. IEEE Computer Society.

[Sha01] E. Shahar. *Tools and Techniques for Verifying Parameterized Systems*. PhD thesis, Faculty of Mathematics and Computer Science, The Weizmann Inst. of Science, Rehovot, Israel, 2001.

[SJ05] Z. Sawa and P. Jančar. Behavioural Equivalences on Finite-State Systems are PTIME-hard. *Computing and Informatics*, 24(5):513–528, 2005.

[Som11] F. Somenzi. CUDD: CU Decision Diagram Package Release 2.4.2. `http://vlsi.colorado.edu/~fabio/CUDD/`, 2011.

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

[TH03] A. Tozawa and M. Hagiya. XML Schema Containment Checking Based on Semi-implicit Techniques. In *Proc. of CIAA*, volume 2759 of *LNCS*, pages 51–61, Berlin, Heidelberg, 2003. Springer Verlag.

[YLB+08] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag.

[YLC+07] H. Yang, O. Lee, C. Calcagno, D. Distefano, and P.W. O'Hearn. On Scalable Shape Analysis. Technical Report RR-07-10, Queen Mary, University of London, 2007.

[ZKR08] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *Proc. of PLDI*, volume 43 of *ACM SIGPLAN Notices*, pages 349–361, New York, NY, USA, 2008. ACM Press.