BRNO UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Intelligent Systems

UNIVERSITY OF GRENOBLE

Jiří Šimáček

HARNESSING FOREST AUTOMATA FOR VERIFICATION OF HEAP MANIPULATING PROGRAMS

VERIFIKACE PROGRAMŮ SE SLOŽITÝMI DATOVÝMI STRUKTURAMI

VÉRIFICATION DE PROGRAMMES AVEC STRUCTURES DE DONNÉES COMPLEXES

Extended abstract of Ph.D. Thesis

Supervisors: prof. Tomáš Vojnar
prof. Yassine Lakhnech
Co-supervisor: dr. Radu Iosif
Opponents:

Presentation date:

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.

The work presented in the 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 **1**
1.1 Goals of the Work 2
1.2 An Overview of the Achieved Results 3
1.3 Plan of the Thesis 6

2 Forest Automata **6**
2.1 Basic Encoding of Heaps 7
2.2 Hierarchical Encoding of Heaps 8
2.3 Future Directions 10

3 Forest Automata-based Verification **11**
3.1 Basic Principles of the Verification Procedure 11
3.2 Future Directions 12

4 Simulations over LTSs and Tree Automata **13**
4.1 Original Algorithm for Computing Simulations 14
4.2 Improved Algorithm for Computing Simulations 14
4.3 Future Directions 14

5 Efficient Inclusion over Tree Automata **15**
5.1 Upward Inclusion Checking 16
5.2 Downward Inclusion Checking 16
5.3 Dealing with Semi-Symbolic Encoding 16
5.4 Future Directions 17

6 A Tree Automata Library **17**
6.1 Existing Libraries 17
6.2 VATA 18
6.3 Future Directions 18

7 Conclusions and Future Directions **19**
7.1 A Summary of the Contributions 19
7.2 Further Directions 20
7.3 Publications and Tools Related to this Work 22

Reference **23**

Curriculum Vitae **28**

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 assu-

rance. 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 Forest Automata

In this section, 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 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.

2.1 Basic Encoding of Heaps

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 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 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.

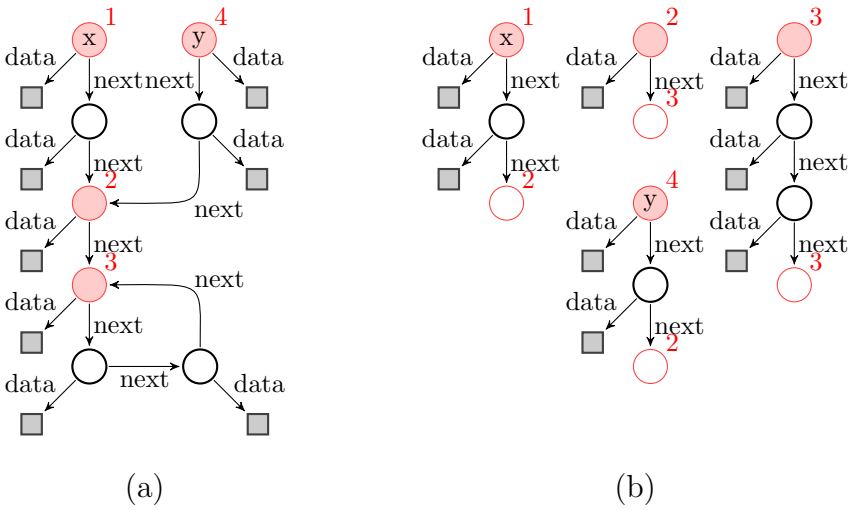


Figure 1: (a) A heap graph with cut-points highlighted in red, (b) the canonical tree decomposition of the heap with `x` ordered before `y`

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. Thus, we represent unions of FA explicitly as *sets of FA* (SFA), which is similar to dealing with disjunctions of conjunctive separation logic formulae. However, as we will see, inclusion on the sets of heaps represented by SFA is still easily decidable.

2.2 Hierarchical Encoding of Heaps

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

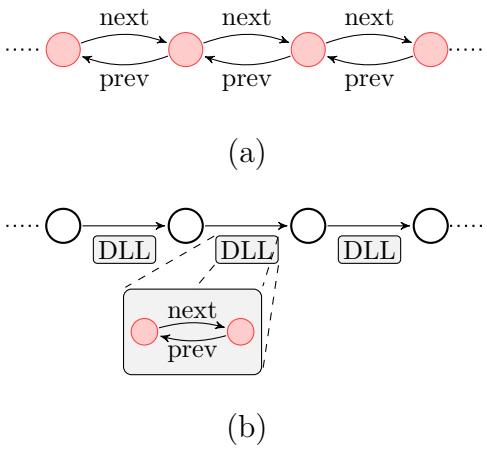


Figure 2: (a) A part of a DLL, (b) a hierarchical encoding of the DLL

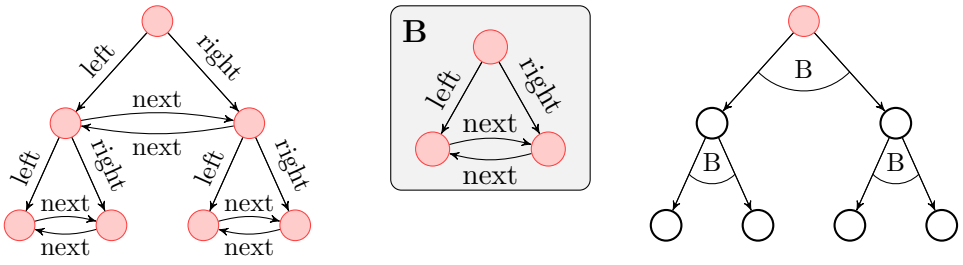
the target vertex of the edge matches the output port.[1] In this way, a set of heap graphs with an unbounded number of cut-points can be transformed into a set of *hierarchical heap graphs* with a bounded number of cut-points at each level of the hierarchy. Figures 2 (a) and (b) illustrate how this 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. The tree with linked brothers from Figure 3 (a) is turned into a hypergraph with binary hyperedges shown in Figure 3 (c) using the box B from Figure 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 (c) encodes it by a hyperedge with one source and *two* target nodes.

Sets of heap hypergraphs corresponding either to the top level of the representation or to boxes of different levels can then be decomposed into (hyper)tree components and represented using *hierarchical FA* whose alphabet can contain nested FA.[2] Intuitively, FA appearing in the alphabet of some superior FA play

[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.

[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.



(a) (b) (c)

Figure 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.

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.

2.3 Future Directions

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 section, is then the problem of computing intersections of FA. Finally, one can also

[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.

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.

3 Forest Automata-based Verification

In this section, 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 the previous section, 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.

3.1 Basic Principles of the Verification Procedure

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, e.g., [BHRV06b]) 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]). 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).

3.2 Future Directions

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.

4 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$^+$08], it has been proposed to replace their use by nodeterministic tree automata (NTA). This, however, brings some problems to be solved. In particular, one needs to be able to perform inclusion checking (in order to see when a fixpoint is reached) and to reduce the of the automata obtained in the computation. Unfortunately, both standard minimisation and inclusion checking algorithms are based on first making the appropriate automata deterministic. Using determinisation as an intermediate step would, however, destroy the advantage of working with usually much smaller NTA. Hence, both of the operations are to be done without determinisation. For checking inclusion, one can use methods based on antichains, possibly combined with simulation as discussed in Section 5 and in Section 6. For reducing the size of the automata, one can use quotienting w.r.t. a suitable, typically simulation-based equivalence relation (see [ABH$^+$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.

4.1 Original Algorithm for Computing Simulations

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.

4.2 Improved Algorithm for Computing Simulations

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.

4.3 Future Directions

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.

5 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 Section 3, and many others) the work in [MPQ11] which deals with the verification of programs manipulating heap structures with data.

In Section 4, 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).

5.1 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.

5.2 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).

5.3 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.

5.4 Future Directions

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.

6 A Tree Automata Library

This section briefly 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 Section 4 or various inclusion checking methods described in [ACH+10] or in Section 5.

The main motivation behind the development of the library is to achieve a better performance of the Forester verification tool presented in Section 3. 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.

6.1 Existing Libraries

Currently, there exist several available tree automata libraries, which are mostly written in high-level languages such as 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.

6.2 VATA

In order to allow researchers focus on developing verification techniques rather than reimplementing and optimising a TA package, we provide VATA[4], 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.

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 Section 3. The semi-symbolic encoding of TA has so far been used mainly for experiments with various newly proposed algorithms for handling TA.

6.3 Future Directions

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)

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

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.

7 Conclusions and Future Directions

Here, we summarise once more the main points and discuss possible future directions of the research from a broader perspective.

7.1 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 Section 6.

7.2 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 Section 3).

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.

7.3 Publications and Tools Related to this Work

The verification technique for programs manipulating complex dynamic data structures and the underlying formalism of forest automata 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 as well as the automatic discovery of nested FA 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[5][6].

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

Reference

[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.

[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.

[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.

[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.

[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.

[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.

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

[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.

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

[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.

[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.

Curriculum Vitae

Personal Data

Name:	Jiří Šimáček
Born:	25 September 1983, Brno, Czech Republic
E-mail:	isimacek@fit.vutbr.cz
Web page:	http://www.fit.vutbr.cz/∼isimacek/

Education

2008 – 2012

- Vysoké učení technické v Brně, Fakulta informačních technologií, Ústav inteligentních systémů (Brno University of Technology)
  - Ph.D. candidate[7]
- Université de Grenoble, Ecole Doctorale, Mathématiques, Sciences et Technologies de l'Information, Informatique (University of Grenoble)
  - Ph.D. candidate[7]

2006 – 2008

- Vysoké učení technické v Brně, Fakulta informačních technologií, Ústav inteligentních systémů (Brno University of Technology)
  - Ing. Information Technology (Czech equivalent of Msc.)

2003 – 2006

- Vysoké učení technické v Brně, Fakulta informačních technologií (Brno University of Technology)
  - Bc. Information Technology (Czech equivalent of BSc.)
- Masarykova Univerzita, Fakulta informatiky (Masaryk University)
  - Bc. Applied Informatics (Czech equivalent of Bsc.)

1995 – 2003

- Gymnázium, Brno, třída Kapitána Jaroše 14

[7]The cotutelle agreement between Brno University of Technology and Université de Grenoble under the supervision of Tomáš Vojnar and Radu Iosif.