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

Ing. Ondřej Lengál

Automata in Infinite-State Formal Verification

Automaty v nekonečně stavové formální verifikaci

Extended Abstract of a Ph.D. Thesis

Supervisor: prof. Ing. Tomáš Vojnar, Ph.D.

Co-Supervisor: Mgr. Lukáš Holík, Ph.D.

Key Words

Antichains, binary decision diagrams, finite automata, heaps, language inclusion, monadic logic, nondeterminism, regular tree model checking, second-order logic, separation logic, shape analysis, simulation, tree automata, formal verification.

Klíčová slova

Antiřetězce, analýza tvaru, binární rozhodovací diagramy, druhořádová logika, haldy, jazyková inkluze, konečný automat, monadická logika, nedeterminismus, regulární stromový model checking, separační logika, simulace, stromový automat, formální verifikace.

The original of the thesis is available in the library of Faculty of Information Technology, Brno University of Technology, Czech Republic.

Contents

1 Introduction

Computer-based systems and technologies keep penetrating still deeper into human lives. The importance of their uninterrupted and correct operation thus keeps growing. Today, computer systems are widely used in the automotive industry, aerospace industry, telecommunication, bank sector, military, etc. An incorrect behaviour of a computer system in some of these environments may cause substantial loses of money, resources, or, in the worst case, even human lives. Even in cases of programs that are not *safety-critical*, errors are often the cause of a negative user experience, which can lead to frustration, and, in an extreme case, even to damage to hardware.

Verification is a process that checks whether a system is *correct* with respect to a provided *specification*. There are two main approaches to verification: bug hunting and formal verification. *Bug hunting* methods focus on finding as many errors as possible in the verified system. This approach includes testing of programs using random inputs while observing their behaviour, dynamic analysis (extrapolation of program's dynamic behaviour), some forms of static analysis (such as detection of errors that match some patterns in the source code), bounded model checking (systematic search of the state space of the program to a limited depth), etc. Bug hunting methods usually cannot guarantee a program's correctness and often find only easily reachable errors.

1.1 Formal Verification

Formal verification is, as opposed to bug-hunting, a technique that attempts to *formally* prove that the verified system is error-free, i.e. formal verification can guarantee that if it does not find an error, there are indeed no errors present in the system. Although the formal verification problem is in general *undecidable*, there are currently various formal verification methods that work well for a large range of classes of programs.

Several properties are often required from formal verification methods. Perhaps the most important of these properties is *soundness*. A method is said to be sound in case it never pronounces a system error-free when the system contains a behaviour that violates the specification. On the other hand, a method is said to be *complete* if it does not produce *spurious counterexamples*, i.e. counterexamples that in fact can never occur in the real system.

1.2 Shape Analysis of Programs Manipulating Heap

One particular class of errors are the ones relating to *memory safety* in programs that use dynamic memory allocation, such as programs manipulating different flavours of lists (e.g. singly/doubly linked, circular, with skip pointers) and trees (e.g. binary trees, trees with root/parent pointers). The area that investigates techniques for dealing with them is called *shape analysis*. Examples of the considered errors are invalid pointer dereference (which may cause a corruption of data values or an abnormal termination of the program) or occurrence of garbage (which may cause the program to deplete the memory available and even affect other programs running on the computer). Dynamic memory is utilised (either directly or indirectly via library calls) in a vast portion of currently produced software. Among the most critical applications that extensively use dynamic memory are kernels of operating system (e.g. Linux) and various standard libraries (e.g. the GNU C library `glibc`).

Because programs manipulating heap are usually infinite-state, a sound analysis technique needs to represent the heap *symbolically*, i.e. represent sets of heaps by different means than enumerating all of their elements. Currently, there are several competing approaches for symbolic heap representation. The first approach is based on the use of formulae of various logics to describe sets of heap configurations. The logics used are separation logic [Rey02, MTLT10, BCC$^+$07, YLB$^+$08, DPV13, CDNQ12, LGQC14], monadic second-order logic [MS01, JJSK97, MPQ11, MQ11], or other [SRW02, ZKR08]. Another approach is based on the use of automata. In this approach, elements of languages of the automata describe configurations of the heap [BHRV06, BBH$^+$11]. The last approach that we will mention is based on graph grammars describing heap graphs [HNR10]. The presented approaches differ in their degree of specialisation for a particular class of data structures, their efficiency, and their level of dependence on user assistance (such as definition of loop invariants or inductive predicates for the considered data structures).

The works that build on *separation logic*, such as [BCC$^+$07, YLB$^+$08, LGQC14], are among the more efficient ones, due to the support for local reasoning provided by the *separating conjunction* (which effectively decomposes the heap into disjoint components so that each can be handled independently of the others, without the need to consider all possible aliasings of their elements). However, most of the techniques based on separation logic are either specialised for some particular data structure—such as singly/doubly linked lists—and even a slight change in the data structure can make the technique unusable (as e.g. in [BCC$^+$07, YLB$^+$08, DPV13]), or they need the user to provide inductive definitions of the used data structures. Moreover, when testing for a fixpoint (which is done to detect whether a newly obtained symbolic representation is subsumed by some already existing one), the analysis needs to check entailment of a pair of separation logic formulae. Entailment procedures have so far been either for considerably limited classes of data structures (e.g. singly linked lists), or quite ad-hoc, based on folding/unfolding inductive predicates in the formulae and trying to obtain a syntactic proof of the entailment. Obviously, this often came with no completeness guarantee. Only recently have there appeared more systematic approaches [IRŠ13, IRV14].

The shape analysis techniques based on automata can address this issue by exploiting the generality of the automata-based representation. Finite tree automata, for instance, have been shown to provide a good balance between efficiency and expressiveness. In particular, the so-called *abstract regular tree model checking* (ARTMC) of heap-manipulating programs [BHRV12] uses a finite tree automaton to describe a set of heaps positioned on a tree backbone (non-tree edges of the heap are represented using regular "routing" expressions describing how the target can be reached from the source using tree edges). Manipulation with the heap is represented using a finite tree transducer and the set of reachable configurations is computed by iteratively applying the transducer on the initial configuration, until a fixpoint is reached. At each step, the obtained symbolic configuration is safely over-approximated using abstraction—which collapses certain states of the automaton—and a fixpoint is detected by standard automata language inclusion testing. The abstraction used is derived automatically during the run of the analysis, using the so-called *counterexample-guided abstraction refinement* (CEGAR) technique. This formalism is able to fully automatically verify even as complex data structures as binary trees with linked leaves, however, it suffers from the inefficiency of the monolithic encoding of the sets of heaps and the transition relation.

Recently, a technique borrowing the best from the worlds of separation logic and ARTMC emerged. This technique, introduced in [HHR$^+$12], is based on the so-called *forest automata*, which are essentially tuples of tree automata where leaves of the trees accepted by one tree automaton can reference roots of the trees accepted by the other tree automata (or by itself). This "non-monolithic" encoding gives a support for local reasoning—heap manipulating operations are executed as simple operations locally on a particular tree automaton, not affecting the other tree automata. Each root of a tree corresponds to a *cut-point* (a node with multiple incoming edges) in the heap graph. Some data structures have an unbounded number of cut-points, e.g. doubly linked lists wherein every internal node is a cut-point. Data structures of this kind cannot be represented in a finite way using this basic formalism; the number of tree components of the forest automata in the analysis would keep growing. The approach therefore uses *hierarchical encoding*, which uses special symbols—called *boxes*, represented using forest automata—to encode sets of subgraphs that contain a cut-point. The technique uses automata abstraction from ARTMC to obtain a sound over-approximation of reachable configurations and accelerate obtaining a fixpoint of the analysis.

1.3 Selected Problems in Shape Analysis

One issue of the techniques described in the previous text is that they often ignore the data component of the represented data structure. This is not always feasible because some data structures, such as binary search trees or skip lists, depend on the data stored inside—in a binary search tree, for example, if a new value is inserted, the ordering relation determines whether the new value is inserted into the left or the right subtree. Examples of works also considering data stored in data structures are [MPQ11, MQ11].

Another interesting problem emerging in the frameworks for shape analysis is the problem of detecting whether the analysis of symbolic executions of a loop has reached a fixpoint. A symbolic execution is an abstract execution of the program that uses the symbolic representation of the program's memory. In this case, the fixpoint is a *closed* representation of the set of reachable configurations of the heap, with *closed* meaning that any new iteration over the body of the loop cannot add anything new to the set. A fixpoint is detected by testing inclusion of the symbolically represented sets of states before and after one more execution of the loop. The analyses based on separation logic perform such a test by checking *entailment* of a pair of formulae describing the heap configurations. On the other hand, in the analyses based on automata, this test corresponds to checking *inclusion of languages* of a pair of automata. Also note that both of these problems are general and used in other settings, such as in deductive verification when deducing whether a precondition of a statement and its semantics imply its postcondition (for entailment), or testing containment of a pair of XML schemas (for tree automata language inclusion), among many others. These problems are theoretically very hard with a discouraging worst case complexity, yet good heuristics can often solve an average case in reasonable time.

An example of such a heuristic is the technique of the so-called *antichains* for checking language inclusion of a pair of nondeterministic finite automata (over finite words or trees). The technique [WDHR06, BHH$^+$08, ACH$^+$10] avoids explicit determinisation of the automata by performing an on-the-fly exploration of the state space. During the exploration, it prunes parts of the state space using a subsumption relation on sets of states of the original automaton (the simplest form of the relation, introduced in [WDHR06], is simple set inclusion). Although language inclusion of a pair of nondeterministic automata has a forbidding worst case complexity—it is a **PSPACE**-complete problem for finite word automata and, even worse, **EXPTIME**-complete problem for finite tree automata—the technique works well for many practical examples.

1.4 Goals of the Thesis

The main goal of this thesis is an improvement of current state of the art in shape analysis. This goal consists of the following three subgoals. The first subgoal is the development of extensions to the shape analysis technique proposed in [HHR$^+$12] that would extend its degree of automation and class of programs it can handle, with a particular focus on data-dependent programs. The second subgoal is an extension and development of new efficient algorithms for testing entailment and validity of selected logics that are used in shape analysis, in particular separation logic and monadic second-order logic. For both of the logics, there exist fragments for which there have been developed efficient translations of decision problems in the logics into finite (tree) automata; such fragments are the particular focus of our attention. For separation logic, we consider the fragment where higher-order inductive predicates correspond to linked lists of many different kinds (singly and doubly linked, circular, nested, . . .), and for monadic second-order logic, we consider its weak fragment of one successor (the so-called weak monadic second-order logic of 1 successor—WS1S). The third subgoal of this thesis is development of techniques for efficient manipulation with finite tree automata, which underlie the previous two subgoals. In particular, the emphasis is placed on the development of algorithms for efficient testing of inclusion over nondeterministic tree automata, and on techniques for manipulating automata with large alphabets.

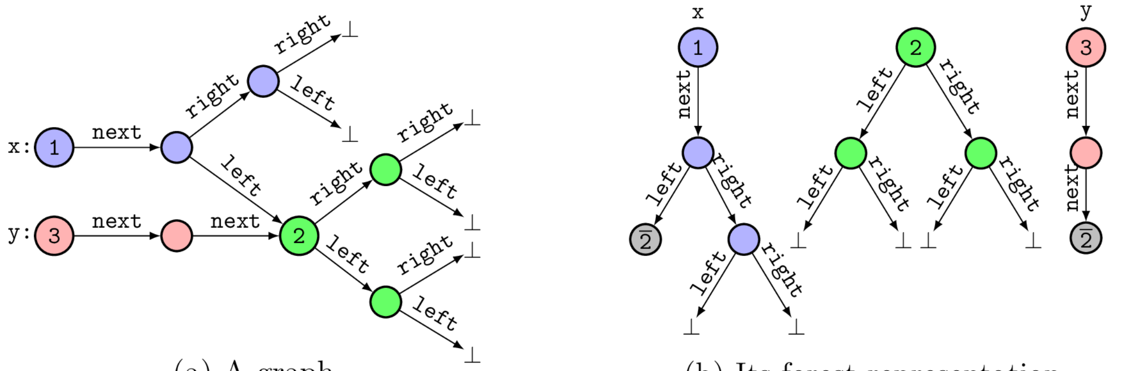
The rest of this text gives an overview of the contributions achived within the areas marked out by the goals of the thesis.

2 Preliminaries

A *ranked alphabet* Σ is a finite 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. 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. A *leaf* of t is a node v which does not have any children, i.e. there is no $i \in \mathbb{N}$ with $vi \in dom(t)$.

A (finite, nondeterministic) *tree automaton* (TA) is a quadruple $\mathcal{A}$ defined as $\mathcal{A} = (Q, \Sigma, \Delta, R)$ where Q is a finite set of *states*, Σ is a ranked alphabet, $R \subseteq Q$ is a set of *root states*, and Δ is a set of *transitions* of the form $q \to a(q_1, \ldots, q_{|a|})$ for $q, q_1, \ldots, q_{|a|} \in Q$ and $a \in \Sigma$. In the special case when $n = 0$, we speak about the so-called *leaf transitions*. For $q \in Q$, $a \in \Sigma$, and $P \subseteq Q$, we use $down_a(q)$ and $down_a(P)$ to denote the set of tuples accessible over a from q and P respectively; formally, $down_a(q) = \{(q_1, \ldots, q_n) \mid q \to a(q_1, \ldots, q_n) \in \Delta\}$ and $down_a(P) = \bigcup_{p \in P} down_a(p)$.

A *run* of $\mathcal{A}$ over a tree t over Σ is a mapping $\rho : dom(t) \to Q$ s.t. for each node $v \in dom(t)$ where $q = \rho(v)$, if $q_i = \rho(vi)$ for $1 \leq i \leq \#t(v)$, then Δ has a transition $q \to t(v)(q_1, \ldots, q_{\#t(v)})$. We write $t \Longrightarrow_\rho q$ to denote that ρ is a run of $\mathcal{A}$ over t s.t. $\rho(root(t)) = q$. We use $t \Longrightarrow q$ to denote that $t \Longrightarrow_\rho q$ for some run ρ. The *language* of a state q is defined by $L(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of $\mathcal{A}$ is defined by $L(\mathcal{A}) = \bigcup_{q \in R} L(q)$.



(a) A graph

(b) Its forest representation

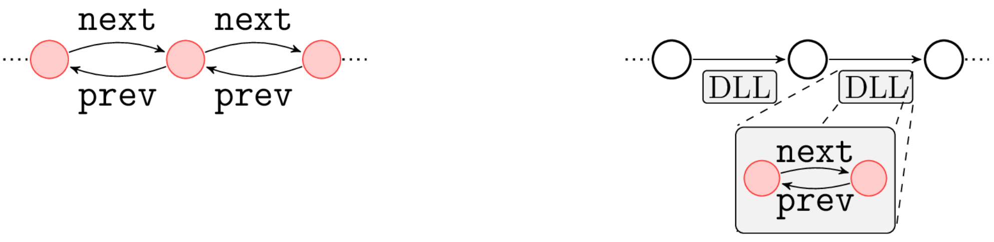
Figure 1: A graph and its forest representation

3 Forest Automata-Based Shape Analysis

In this section, we focus on our contributions in extending the forest automata-based framework presented in [HHR$^+$12]. The main concept of the symbolic representation used in the framework is the so-called *forest decomposition* of a heap graph, which is performed as follows: First, the *cut-points* of the graph are identified; a cut-point is a node that is either referenced by a program variable or is a target of multiple edges. Every cut-point is then taken as the root of a (cut-point-free) tree component whose leaves are either nodes with no outgoing edges, or other cut-points. The heap graph is split into the tree components, which are then canonically ordered according to the order in which their roots were visited in a depth-first search (DFS) through the graph, when starting from program variables. In the tree components, any leaf that corresponds to a cut-point numbered with c during the DFS is changed into an explicit reference to the cut-point number c, written as $\overline{c}$. See Figure 1 for an illustration of the forest decomposition of a heap graph.

To represent a (potentially infinite) *set* of heaps $H = \{h_1, h_2, \dots\}$ with the same number n of cut-points, we decompose all heaps of H into forests and for every position $1 \leq i \leq n$, we then collect the i-th components of all forests into the set $H[i] = \{h_1[i], h_2[i], \dots\}$. The set $H[i]$ can be represented using a TA $\mathcal{A}[i]$ and the whole set of heaps H can be represented by a tuple of TAs $\mathcal{A}[1], \dots, \mathcal{A}[n]$, called a *forest automaton* (FA).

An FA of the simple structure presented above cannot be used as a representation of data structures that have an unbounded number of cut-points—such as doubly linked lists (DLLs) or trees with parent pointers, where every internal node is a cut-point—and the analysis would need an infinite number of FAs to represent a set of all instances of these data structures. In order to be able to represent them using finitary means, the forest automata framework allows the use of the so-called *boxes*. Boxes are FAs that are used as symbols of another, *higher level* FA. In this FA, they represent a (complex) subgraph



(a) A DLL (b) Its hierarchical encoding

Figure 2: A DLL and its hierarchical encoding

using a single symbol. Intuitively, the task of boxes is to decrease the in-degree of cut-points in a graph—when the in-degree of a node drops to one (and the node is not referenced by a program variable), the node is no longer a cut-point and can be represented by an ordinary state in a TA. In this way, it is possible to represent an over-approximation of all reachable configurations of a program using forest automata with a bounded number of tree components. See Figure 2 for an example of a use of a box in an encoding of a DLL.

Alongside the notion of FAs, [HHR$^+$12] also proposed a shape analysis that uses FAs and is based on the framework of *abstract interpretation* [CC77]. For each program line, a set of forest automata is used to represent the set of memory configurations reachable at a given line. The program is symbolically executed on this representation in such a way that each program statement is mapped to an *abstract transformer* that simulates execution of the statement on the symbolic representation (and also checks whether an error has been encountered). The symbolic execution examines all branches of the program until no new symbolic states can be found on the branches and a fixpoint is obtained. Because, as mentioned earlier, programs manipulating heap are usually infinite-state, the *widening* operator is used to provide a sound over-approximation of the set of reachable configurations. This operator is based on automata *abstraction* borrowed from ARTMC. For a given forest automaton, abstraction collapses some states of the TAs of the FA (for each TA separately), trying to introduce loops into the TAs to obtain TAs accepting an infinite (regular) tree language that over-approximates the original one and, in turn, a forest automaton representing an infinite set of heaps, again over-approximating the original one.

3.1 Learning of Boxes

The shape analysis described in the previous section relied on the user to provide a suitable set of boxes (the subgraphs to be folded into automata symbols). This means that the user needed to provide the analysis with a forest automata-based description of those data structures used in the program that have an unbounded number of cut-points. As we strive for a *push-button* analysis, such an approach is naturally not feasible. To address this issue, we

Table 1: Results of experiments with Forester and box learning

Example	Forester	boxes	Predator	Example	Forester	boxes	Predator
SLL (delete)	0.04		0.04	DLL (reverse)	0.06	1 / 1	0.04
SLL (bubblesort)	0.04		0.03	DLL (insert)	0.07	1 / 1	0.05
SLL (mergesort)	0.15		0.10	DLL (insertsort1)	0.40	1 / 1	0.11
SLL (insertsort)	0.05		0.04	DLL (insertsort2)	0.12	1 / 1	0.05
SLL (reverse)	0.03		0.03	DLL of CDLLs	1.25	8 / 7	0.22
SLL+head	0.05		0.03	DLL+subdata	0.09	- / 2	T
SLL of 0/1 SLLs	0.03		0.11	CDLL	0.04	1 / 1	0.04
SLL$_{\text{Lin}}$	0.03		0.03	tree	0.14		Err
SLL of CSLLs	0.74	4 / 4	0.12	tree+parents	0.21	2 / 2	T
SLL of 2CDLLs$_{\text{Lin}}$	0.17	14 / 5	0.25	tree+stack	0.08		Err
skip list$_2$	0.42	- / 4	T	tree (DSW)	0.40		Err
skip list$_3$	9.14	- / 7	T	tree of CSLLs	0.42	- / 4	Err

propose an extension of the approach where the boxes are inferred automatically during a run of the analysis using a technique that we call *box learning*.

The basic principle of our box learning is to identify suitable subgraphs of the FA-represented graphs that contain at least one join, and when they are enclosed—or, as we say, *folded*—into a box, the in-degree of the join decreases. There are, of course, many ways to select the above mentioned subgraphs to be used as boxes. To choose among them, we propose several criteria that we found useful in a number of experiments. Most importantly, the boxes must be *reusable* in order to allow eliminating as many joins as possible. The general strategy here is to choose boxes that are *simple* and *small* since these are more likely to correspond to graph patterns that appear repeatedly in typical data structures. For instance, in the already mentioned case of DLLs in Figure 2, it is enough to use a box enclosing a single pair of `next`/`prev` links. On the other hand, too simple boxes are sometimes not useful either, because they fail to hide a join in the graphs.

Further, we propose a way how box learning can be efficiently integrated into the main analysis loop. We always try to identify which subgraphs of the graphs represented by a given FA could be folded into a box, followed by looking into the so-far built database of boxes whether such a box has already been introduced or not. This approach has the advantage that it allows one to use simple language inclusion checks for *approximate box folding* that substitutes a subgraph with a box from the database having a *larger* language, thus over-approximating the set of graphs represented by a given FA. This approach sometimes greatly accelerates the computation. Finally, to further improve the efficiency, we interleave the process of box learning with the *automata abstraction* into a single iterative process. In addition, we propose an FA-specific improvement of the basic automata abstraction that *accelerates the abstraction* of an FA using components of other FAs. Intuitively, it lets the abstraction synthesise an invariant faster by allowing it to combine information coming from different branches of the symbolic computation.

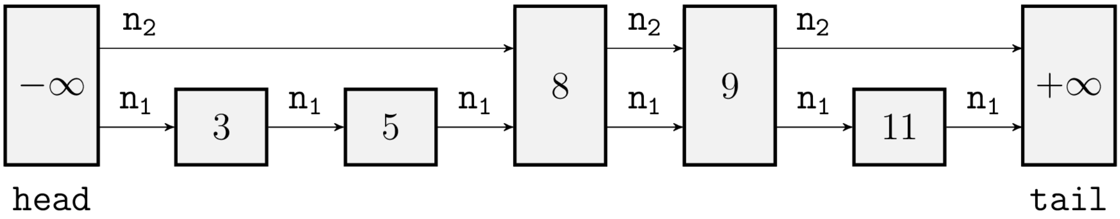


Figure 3: An example of a 2-level skip list

3.1.1 Experimental Results

We compared an implemenation of our techniques (in a tool named Forester) with Predator [DPV13], which uses a graph-based memory representation inspired by separation logic with list predicates. In the experiments, we considered programs with various types of lists (singly and doubly linked, cyclic, nested, with skip pointers), trees, and their combinations. We used a data-oblivious modification of a skip list that remembers the window it is inserting into by an explicit pointer. The programs were checked for memory safety.

Table 1 gives running times in seconds (the average of 10 executions) of the tools on our case studies. The table further contains the column "boxes" where the value "X/Y" means that X manually created boxes were provided to the analysis that did not use learning while Y boxes were learnt when the box learning procedure was enabled. The value "-" of X means that we did not run the example with manually constructed boxes since their construction was too tedious. If user-defined boxes are given to Forester in advance, the speedup is usually negligible, with the exception of "DLL of CDLLs" and "SLL of CSLLs", where it is up to 7 times. In a majority of cases, the learnt boxes were the same as the ones created manually. In some cases, though, the learning algorithm managed to find a smaller set of more elaborate boxes.

Note that the performance of Forester in the considered experiments is indeed comparable with that of Predator even though Forester can handle much more general data structures, such as a data-oblivious modification of two and three level skip lists (a modification where the shape invariant of a skip list does not rely on the fact that the list is ordered—our extension to the standard data-dependent skip lists is described in Section 3.2).

3.2 An Extension to Programs with Ordered Data

We also extended the forest automata-based shape analysis to programs that rely on ordered data, such as programs with binary search trees (BSTs) or skip lists (see Figure 3). Technically, our extension express relationships between data elements associated with nodes of the heap graph by two classes of constraints. *Local data constraints* are associated with transitions of TAs and capture relationships between data of neighbouring nodes in a heap graph; they can be used e.g. to represent ordering internal to some structure such

Table 2: Results of the experiments with the data extension of Forester

Example	time	Example	time	Example	time	Example	time
SLL insert	0.06	DLL insert	0.14	SL_2 insert	9.65	BST insert	6.87
SLL delete	0.08	DLL delete	0.38	SL_2 delete	10.14	BST delete	15.00
SLL reverse	0.07	DLL reverse	0.16	SL_3 insert	56.99	BST left rotate	7.35
SLL bubblesort	0.13	DLL bubblesort	0.39	SL_3 delete	57.35	BST right rotate	6.25
SLL insertsort	0.10	DLL insertsort	0.43				

as a binary search tree. *Global data constraints* are associated with states of TAs (even states of *different* TAs) and capture relationships between data in distant parts of the heap. In order to obtain a powerful analysis based on such extended FAs, the entire analysis machinery must be redesigned, including a need to develop mechanisms for propagating data constraints through FAs, to adapt the abstraction mechanisms of ARTMC, to develop a new inclusion check between extended FAs, and to define extended abstract transformers.

3.2.1 Experimental Results

The presented approach has been implemented as a further extension of the Forester tool. We have applied the tool to verification of data properties, notably sortedness, of sequential programs with data structures, like various forms of singly and doubly linked lists (SLLs and DLLs), binary search trees (BSTs), and even two and three level skip lists (SL_2 and SL_3). Results of the experiments are summarised in Table 2, which gives running times in seconds of Forester on the case studies. The experiments confirm that our approach is not only fully automated and rather general, but also quite efficient, outperforming many previously known approaches even though they are not of the same level of automation or generality. In the case of skip lists, our analysis is the first fully-automated shape analysis which is able to handle fully-fledged skip lists.

4 Deciding Logics with Automata

In this section, we describe our automata-based decision procedures for the following two logics: (a) separation logic with list predicates and (b) weak monadic second order logic of one successor (WS1S).

4.1 Separation Logic with List Predicates

Separation logic (SL) [Rey02] is a formalism for shape analysis complementary to the formalism of forest automata presented in the previous section. It offers both high expressiveness and scalability, the latter being due to its support of *compositional reasoning* based on the separating conjunction $*$ and the frame rule, which states that if a Hoare triple $\{\phi\}P\{\psi\}$ holds and P does not alter free variables in σ, then $\{\phi*\sigma\}P\{\psi*\sigma\}$ holds too. So, when reasoning about P, one has to manipulate only specifications for the heap region altered by P.

In the work presented in this chapter, we focus on a fragment of SL with inductive definitions that allows one to specify program configurations (heaps) containing finite nestings of various kinds of linked lists (acyclic or cyclic, singly or doubly linked, skip lists, etc.), which are common in practice. This fragment contains formulae of the form $\exists \overrightarrow{X} : \Pi \wedge \Sigma$ where X is a set of variables, Π is a conjunction of (dis-)equalities, and Σ is a set of *spatial atoms* connected by the separating conjunction. Spatial atoms can be *points-to atoms*, which describe values of pointer fields of a given heap location, or *inductively defined predicates*, which describe data structures of an unbounded size. We propose a novel (semi-)decision procedure for checking the validity of entailments of the form $\varphi \Rightarrow \psi$ where φ may contain existential quantifiers and ψ is a quantifier-free formula. Such a decision procedure can be used in Hoare-style reasoning to check inductive invariants, but also in program analysis frameworks to decide termination of fixpoint computations. As usual, checking entailments of the form $\bigvee_i \varphi_i \Rightarrow \bigvee_j \psi_j$ can be soundly reduced to checking that for each i there exists j such that $\varphi_i \Rightarrow \psi_j$.

The key insight of our decision procedure is an idea to use the semantics of the separating conjunction to decompose the problem of checking $\varphi \Rightarrow \psi$ to the problem of checking a set of simpler entailments of the form $\varphi_i \Rightarrow P_i(\ldots)$—where the right-hand side is an inductively-defined predicate—and discharging these simpler queries one by one using reduction to the tree automata membership problem. The algorithm proceeds in the three steps described further.

First, φ and ψ are normalised by adding (dis-)equalities implied by their pure parts together with the axioms for equality and the semantics of the separating conjunction. Moreover, empty list predicates are removed at this point and the left-hand side of the entailment is split into a set of disjoint subformulae, one for each atom of the right-hand side. Second, every points-to atom of the right-hand side is matched with a corresponding points-to atom in the left-hand side. Finally, for every remaining inductive predicate $P_i(\ldots)$ in the right-hand side and the corresponding subformula φ_i in the left-hand side, the entailment $\varphi_i \Rightarrow P_i(\ldots)$ is discharged in the following way: The formula φ_i is translated into a tree $T[\varphi_i]$ that represents the spanning tree of the graph described by the formula, extended with routing expressions describing relations between nodes that had an edge between them in the graph but miss an edge in the spanning tree. For the other side, the predicate P_i is translated into a tree automaton $\mathcal{A}[P_i(\ldots)]$ encoding trees of all possible unfoldings of the predicate $P_i(\ldots)$. The entailment $\varphi_i \Rightarrow P_i(\ldots)$ holds iff $T[\varphi_i] \in \mathcal{A}[P_i(\ldots)]$.

Our procedure is complete for formulae speaking about non-nested singly as well as doubly linked lists. Moreover, it runs in polynomial time modulo an oracle for deciding validity of a Boolean formula. The procedure is incomplete for nested list structures due to not considering all possible ways in

Table 3: Running Spen on entailments between formulae and atoms.

φ_2	`nll`			`nlcl`			`skl`$_3$			`dll`		
φ_1	`tc1`	`tc2`	`tc3`	`tc1`	`tc2`	`tc3`	`tc1`	`tc2`	`tc3`	`tc1`	`tc2`	`tc3`
Time [ms]	344	335	319	318	316	317	334	349	326	358	324	322
Status	vld	vld	inv	vld	vld	inv	vld	vld	inv	vld	vld	inv
Size of $\mathcal{A}[\varphi_2]$	6/17			6/15			80/193			9/16		
Size of $T[\varphi_1]$	7/7	7/7	6/7	10/9	7/7	6/6	7/7	8/8	6/6	7/7	7/7	5/5

which targets of inner pointer fields of nested list predicates can be aliased. The construction can be easily extended to become complete even in such cases, but then it becomes exponential in the size of the inductive predicate (which still remains acceptable in practice).

4.1.1 Experimental Results

We implemented our decision procedure in a solver called Spen (SeParation logic ENtailment) and applied it to entailment problems $\varphi_1 \Rightarrow \varphi_2$ that use various recursive predicates. First, we considered the benchmark provided in [PR11], which uses only the `ls` predicate. This benchmark has been used in the `ls` division of the first competition of Separation Logic solvers, SL-COMP'14. It consists of 292 problems and Spen solved the full benchmark in less than 8 seconds (CPU time), which is the second time of the division; the winner of the division was a specialized solver for the `ls` predicate, Asterix [PR13], which spent less than 4 seconds on the benchmark.

The TA for `ls` is quite small, and so the above experiments did not evaluate thoroughly the performance of our procedure for checking entailments between formulae and atoms. For that, we further considered the experiments listed in Table 3. The full benchmark includes the 43 problems of the division "fixed definitions" of SL-COMP'14. The entailment problems are extracted from verification conditions of operations like adding or deleting an element at the start, in the middle, or at the end of various kinds of list segments. Table 3 gives for each example the running time, the valid/invalid status, and the size of the tree encoding (in the form "nodes/edges") and TA (in the form "states/transitions") for φ_1 and φ_2, respectively. Spen was the winner in this division of SL-COMP'14 (in front of [BGP12, CDNQ12]) and it was the only tool that solved all problems of this division.

4.2 WS1S

Weak monadic second-order logic of one successor (WS1S) is a powerful, concise, and decidable logic for describing regular properties of finite words. Despite its **NONELEMENTARY** worst case complexity, it has been shown useful in numerous applications. Most of the successful applications were due to the tool MONA [EKM98], which implements a finite automata-based decision procedure for WS1S and WS2S (a generalisation of WS1S to finite binary trees). Despite many optimisations implemented in MONA, the worst case complexity of the problem sometimes strikes back. Authors of methods using the translation of their problem to WS1S/WS2S are then forced to either find workarounds to circumvent the complexity blowup, such as in [MQ11], or, often restricting the input of their approach, give up translating to WS1S/WS2S altogether [WMK11].

The decision procedure of MONA works with deterministic automata; it uses determinisation extensively and relies on minimisation of deterministic automata to suppress the complexity blow-up. The worst case exponential complexity of determinisation, however, often significantly harms the performance of the tool. Recent works on efficient methods for handling nondeterministic automata suggest a way of alleviating this problem, in particular works on efficient testing of language inclusion and universality of finite automata [WDHR06, ACH$^+$10] and size reduction [ABH$^+$08] based on a simulation relation. Handling nondeterministic automata using these methods, while avoiding determinisation, has been shown to provide great efficiency improvements in [BHH$^+$08] (ARTMC) and also [HHR$^+$12] (shape analysis). In this chapter, we present a work that makes a major step towards building the entire decision procedure of WS1S on nondeterministic automata using similar techniques. We propose a generalisation of the antichain algorithms of [WDHR06] that addresses the main bottleneck of the automata-based decision procedure for WS1S, which is also the source of its **NONELEMENTARY** complexity: elimination of alternating quantifiers on the automata level.

More concretely, the automata-based decision procedure translates the input WS1S formula into a finite word automaton such that its language represents exactly all models of the formula. The automaton is built in a bottom-up manner according to the structure of the formula, starting with predefined atomic automata for literals and applying a corresponding automata operation for every logical connective and quantifier ($\wedge, \vee, \neg, \exists$). The cause of the **NONELEMENTARY** complexity of the procedure can be explained on an example formula of the form $\varphi' = \forall X_m \exists X_{m-1} \ldots \forall X_2 \exists X_1 : \varphi_0$. The universal quantifiers are first replaced by negation and existential quantification, which results in $\varphi = \neg\exists X_m \neg\exists X_{m-1} \ldots \neg\exists X_2 \neg\exists X_1 : \varphi_0$. The algorithm then builds a sequence of automata for the sub-formulae $\varphi_0, \varphi_0^\sharp, \ldots, \varphi_{m-1}, \varphi_{m-1}^\sharp, \varphi_m = \varphi$ of φ, where for $0 \leq i < m$, we have $\varphi_i^\sharp = \exists X_{i+1} : \varphi_i$ and $\varphi_{i+1} = \neg\varphi_i^\sharp$. Every automaton in the sequence is created from the previous one by applying the automata operations corresponding to negation or elimination of the existential quantifier, the latter of which may introduce nondeterminism. Negation applied on a nondeterministic automaton may then yield an exponential blowup: given an automaton for ψ, the automaton for $\neg\psi$ is constructed by the classical automata-theoretic construction consisting of determinisation by the subset construction followed by swapping of the sets of final and non-final states. The subset construction is exponential in the worst case. The worst case complexity of the procedure run on φ is then a tower of exponentials with one level for every quantifier alternation in φ; note that we cannot do much better—this **NONELEMENTARY** complexity is an inherent property of the problem.

Our algorithm works on a ground formula φ in the prenex form and is an optimisation of the construction presented in the previous paragraph. We use the following property of the construction: φ is valid iff $\mathcal{A}_m$, the automaton representing the formula $\varphi_m = \varphi$, contains an initial state that is also final, i.e. $I_m \cap F_m \neq \emptyset$ for the set of initial states I_m and the set of final states F_m. The automaton for the matrix φ_0 is constructed in the standard manner. For processing the prefix $\neg\exists X_m \neg\exists X_{m-1} \ldots \neg\exists X_2 \neg\exists X_1$, we perform the sequence of projections and complementations (corresponding to existential quantifiers and negations) symbolically and compute only the sets I_m and F_m of $\mathcal{A}_m$.

Computing the set I_m is easy: it is simply an m-level hierarchy of singleton sets $\{\{\cdots\{I_0\}\cdots\}\}$ (this follows from subset construction and the fact that neither projection nor complementation change initial states of automata). On the other hand, computing the set F_m is more difficult due to two reasons: (i) projection can make some predecessors of final states also final, and (ii) after determinisation and complementation, the final states of the resulting automaton are "swapped." Despite these difficulties, we developed an efficient symbolic representation of sets of final states using a hierarchy of upward and downward closed sets, and also algorithms for computation of predecessors of the sets using the symbolic representation and testing intersection of the sets with the set of initial states. In addition to this, we further proposed an approach for testing subsumption between symbolically represented sets, which is a generalisation of the antichain approach from the domain of testing universality and inclusion of finite automata. This subsumption relation is used for pruning the symbolic representations of states on all levels of the hierarchy.

4.2.1 Experimental Results

We implemented a prototype of the above presented approach in the tool `dWiNA` and evaluated it in a benchmark of both practical and generated examples. The practical formulae for our experiments were obtained from the shape analysis of [MQ11]; the results are shown in Table 4a. We measure the time of runs of the tools for processing only the prefix of the formulae. We can observe that w.r.t. the speed, we get comparable results; in some cases `dWiNA` is slower than MONA, which we attribute to the fact that our prototype

Table 4: Results of experiments for our WS1S decision procedure

a) Results for practical formulae

Benchmark	Time [s]		Space [states]	
	MONA	dWiNA	MONA	dWiNA
reverse-before-loop	0.01	0.01	179	47
insert-in-loop	0.01	0.01	463	110
bubblesort-else	0.01	0.01	1 285	271
reverse-in-loop	0.02	0.02	1 311	274
bubblesort-if-else	0.02	0.23	4 260	1 040
bubblesort-if-if	0.12	1.14	8 390	2 065

b) Results for generated formulae

	Time [s]		Space [states]	
k	MONA	dWiNA	MONA	dWiNA
1	0.11	0.01	10 718	39
2	0.20	0.01	25 517	44
3	0.57	0.01	60 924	50
4	1.79	0.02	145 765	58
5	4.98	0.02	349 314	70
6	∞	0.47	∞	90

implementation is, when compared with MONA, quite immature. Regarding space, we compare the sum of the number of states of all automata generated by MONA when processing the prefix of φ with the number of symbolic terms generated by dWiNA for processing the same. We can observe a significant reduction in the generated state space.

To better evaluate the scalability of our approach, we created several parameterised families of WS1S formulae. In Table 4b, we give the results for one of the families where the basic formula expresses existence of an ascending chain of n sets ordered w.r.t. $\subset$. The parameter k stands for the number of alternations in the prefix of the formulae.

5 Efficient Techniques for Manipulating Nondeterministic Tree Automata

Recently, there has been notable progress in the development of algorithms for efficient manipulation of nondeterministic TAs, more specifically, in solving the crucial problems of automata reduction [ABH$^+$08] and of checking language inclusion [TH03, BHH$^+$08, ACH$^+$10]. As shown e.g. in [BHH$^+$08], replacing deterministic automata by nondeterministic ones can—in combination with the new methods for handling TAs—lead to great efficiency gains.

5.1 Downward Inclusion Checking

In this section, we propose a new algorithm for language inclusion checking of TAs that turns out to significantly outperform the existing algorithms in most of our experiments. The classic textbook algorithm for checking inclusion $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$ between two TAs $\mathcal{A}_S$ (Small) and $\mathcal{A}_B$ (Big) first bottom-up determinises $\mathcal{A}_B$, computes the complement automaton $\overline{\mathcal{A}_B}$ of $\mathcal{A}_B$ (the states, called *macrostates*, of which are sets of states of $\mathcal{A}_B$), and then checks language emptiness of the product automaton accepting $L(\mathcal{A}_S) \cap L(\overline{\mathcal{A}_B})$. This approach has been optimised in [TH03, BHH$^+$08, ACH$^+$10] by avoiding 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 its states and checking language emptiness on the fly. The optimised algorithm is based on starting from the leaf states of both automata and maintaining a set of reachable pairs (q_S, P_B) where q_S is a state of $\mathcal{A}_S$ and P_B is a set of states of $\mathcal{A}_B$. New pairs (q_S, P_B) are generated by taking a tuple of states $(q_1, \ldots, q_n)$ such that every q_i appears in some reachable pair (q_i, P_i) and q_S is a bottom-up *post* of the tuple in $\mathcal{A}_S$ over some symbol a. The set P_B is then obtained as the bottom-up a-post in $\mathcal{A}_B$ of all tuples in $P_1 \times \cdots \times P_n$. In case q_S is a root state and P_B, on the other hand, contains no root state, the algorithm terminates with the answer $L(\mathcal{A}_S) \not\subseteq L(\mathcal{A}_B)$ (this corresponds to finding a witness of non-emptiness of the set $L(\mathcal{A}_S) \cap L(\overline{\mathcal{A}_B})$). If no new pair can be generated, the algorithm concludes that $L(\mathcal{A}_S) \subseteq L(\mathcal{A}_B)$.

The particular optimisation used in [TH03, BHH$^+$08, ACH$^+$10], called the *antichain* principle, is based on removing from the set of reachable pairs those pairs (q_S, P_B) for which there is already a reachable pair (q_S, P'_B) in the set, with $P'_B \subseteq P_B$. The argument why this pruning is correct is that P'_B has a higher chance to generate a set of states that contains no root state. On the other hand, for every set of states reachable from P'_B, there will be a corresponding larger (w.r.t. inclusion) set of states reachable from P_B, so if the set reachable from P'_B contains a root state r, the set reachable from P_B will also contain r. This can be even more optimized by the approach of [ACH$^+$10], which uses the upward simulation relation to weaken the conditions for removing a pair from the set of reachable states. The mentioned optimisations in practice often prove or refute inclusion by constructing a small part of the product automaton only. We denote these algorithms as *upward* algorithms to reflect the direction in which they traverse automata $\mathcal{A}_S$ and $\mathcal{A}_B$.

The upward algorithms are sufficiently efficient in many practical cases. However, they have two drawbacks: (i) When generating the bottom-up post-image of a set $\mathcal{S}$ of macrostates (which are sets of states of $\mathcal{A}_B$), all possible n-tuples of states from all possible products $S_1 \times \ldots \times S_n$ where $S_i \in \mathcal{S}$ need to be enumerated (an optimisation of this was presented in [LŠV12]). (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.

The alternative *downward* approach to checking TA language inclusion was first proposed in [HVP05] in the context of subtyping of XML types. The inclusion 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 that 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).

Our algorithm uses the following theorem. In the theorem, we conveniently exploit the notion of *choice functions*. Given $P_B \subseteq Q_B$ and $a \in \Sigma$, such that $\#a = n \geq 1$, we denote by $cf(P_B, a)$ the set of all choice functions f that assign an index i, for $1 \leq i \leq n$, to all n-tuples $(q_1, \ldots, q_n) \in Q_B^n$ for which there exists a state in P_B that can make a top-down transition over a to $(q_1, \ldots, q_n)$; formally, $cf(P_B, a) = \{f \mid f : down_a(P_B) \to \{1, \ldots, \#a\}\}$.

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

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

The main idea of our algorithm is to start from a pair (r_S, R_B), where $r_S \in R_S$, and use a DFS to traverse the pairs in the *And-Or* tree induced by recursive application of Theorem 1, where *And* nodes correspond to the universal quantification $\forall f \in cf(P_B, a)$ and *Or* nodes correspond to the existential quantification $\exists 1 \leq i \leq \#a$. When traversing the tree, a branch is cut in the case a pair already appearing on the branch is encountered.

We developed several optimisations of the basic algorithm: (1) We cut a branch already in the case when we encounter a pair (q_S, P_B) such that there is a pair (q_S, P'_B) for $P'_B \subseteq P_B$ on the branch (if $L(q_S) \subseteq L(P'_B)$, then it will also hold that $L(q_S) \subseteq L(P_B)$). (2) The algorithm keeps a cache *NN* for pairs (q_S, P'_B) where it has been proved that $L(q_S) \not\subseteq L(P'_B)$. If the algorithm encounters an element (q_S, P'_B) from *NN*, it can also cut the current branch by using the cached result. Moreover, this can be done even if it encounters an element (q_S, P_B) where $P_B \subseteq P'_B$, following the reasoning that if $L(q_S) \not\subseteq L(P'_B)$, then also $L(q_S) \not\subseteq L(P_B)$. The cache *NN* can be maintained as an *antichain*, i.e. to contain for every state q_S of $\mathcal{A}_S$ only those sets P'_B that are maximal w.r.t. set inclusion. (3) Similar to the previous optimisation, the algorithm also keeps a cache *IN* (again as an antichain, but keeping minimal elements in this case) for pairs (q_S, P'_B) where it has been proved that $L(q_S) \subseteq$

Table 5: Results of the experiments with downward inclusion checking

Algorithm	All pairs		$L(\mathcal{A}) \not\subseteq L(\mathcal{B})$		$L(\mathcal{A}) \subseteq L(\mathcal{B})$	
	Winner	Timeouts	Winner	Timeouts	Winner	Timeouts
`down`	36.35 %	32.51 %	39.85 %	26.01 %	0.00 %	90.80 %
`down+s`	4.15 %	18.27 %	0.00 %	20.31 %	47.28 %	0.00 %
`down-opt`	32.20 %	32.51 %	35.30 %	26.01 %	0.00 %	90.80 %
`down-opt+s`	3.15 %	18.27 %	0.00 %	20.31 %	35.87 %	0.00 %
`up`	24.14 %	0.00 %	24.84 %	0.00 %	16.85 %	0.00 %
`up+s`	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %

$L(P'_B)$. If the algorithm encounters an element (q_S, P_B) such that there is $(q_S, P'_B) \in IN$ for $P'_B \supseteq P_B$, it can also cut the current branch by using the cached result (if $L(q_S) \subseteq L(P'_B)$, then it will also hold that $L(q_S) \subseteq L(P_B)$). (4) Finally, we use the following three optimisations that employ a pre-computed downward simulation relation $\preceq$ on the states of $\mathcal{A}_S$ and $\mathcal{A}_B$ and rely on the fact that $\preceq$ is an under-approximation of language inclusion. First, we weaken the set inclusion tests $X \subseteq Y$ for sets of states X, Y used above to the tests $X \preceq^{\forall\exists} Y$, where $X \preceq^{\forall\exists} Y$ holds iff $\forall x \in X \, \exists y \in Y : x \preceq y$. Second, every pair (q_S, P_B) encountered is minimised in such a way that P_B contains no two distinct states r, s such that $r \preceq s$ by removing all simulated states. Third, we cut a branch also if we find a pair (q_S, P_B) such that $\{q_S\} \preceq^{\forall\exists} P_B$.

5.1.1 Experimental Results

Table 5 shows our experimental results. The row `down` denotes the implementation of our algorithm that uses optimisations 1 and 2 from the previous section. The row `down+s` is the extension of `down` with optimisation 4 (using the maximum downward simulation, `down-opt` is the extension of `down` with optimisation 3, and `down-opt+s` contains all optimisations. The row `up` denotes the implementation of the upward inclusion checking algorithm of [BHH$^+$08], and its modification that uses the maximum upward simulation parameterised by identity (proposed in [ACH$^+$10]) is marked as `up+s`.

We ran our experiments on almost 2 000 TA pairs of different sizes (ranging from 50 to 1 000 states) with the timeout set to 30 seconds. The table compares the methods according to the percentage of the cases in which they were the fastest when checking inclusion on the same automata pair, and also according to the percentage of timeouts. The set of results in the column labelled with "All pairs" contains data for all pairs, the results below "$L(\mathcal{A}) \not\subseteq L(\mathcal{B})$" are for the pairs $\mathcal{A}, \mathcal{B}$ where the inclusion does not hold, and the column under "$L(\mathcal{A}) \subseteq L(\mathcal{B})$" reports on the cases where the inclusion holds. The results show that the downward approaches are clearly dominating in significantly more of our test cases. On the other hand, it can be observed that for some particular cases, the more complex structure of the downward algorithms (which resembles an *And-Or* tree) causes an unmanageable state explosion and the algorithms timeout (in contrast to the upward algorithms, which always, though often slowly, terminate).

5.2 Semi-Symbolic Encoding

Certain important applications of TAs, such as formal verification of programs with complex dynamic data structures [BHRV12] or decision procedures of logics such as WSkS (see Section 4.2), require TAs with very large alphabets. Here, the common choice is to use the TA library of MONA [KMS02] to represent transitions of TAs symbolically using *multi-terminal binary decision diagrams*. The encoding of MONA is, however, restricted to *deterministic* automata only. This implies a necessity of immediate determinisation after each operation over TAs that introduces nondeterminism and may, in turn, easily lead to a state space explosion. As a way to overcome this issue, we developed a semi-symbolic representation of *nondeterministic* TAs that generalises the one used by MONA, and we develop algorithms implementing the basic operations on TA: such as union, intersection, removal of top-down and bottom-up unreachable states, as well as more involved algorithms for computing simulations and for checking language inclusion (using simulations and antichains to optimise it) over the proposed representation.

5.3 A Tree Automata Library

The techniques presented in the previous text, as well as many other formal verification techniques, rely on an efficient underlying implementation of TAs, and their success can be hindered by a poor performance of a naïve TA implementation. Currently, there exist several available TA libraries; they are, however, 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 that require the available processing power be utilised as efficiently as possible. An exception from these libraries is MONA but, alas, it supports binary deterministic TAs only. At the same time, it turns out that determinisation is often a very significant bottleneck of using TAs, and a lot of effort has therefore been invested into developing efficient algorithms for handling nondeterministic TAs 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 developed Vata, an easy-to-use open source library for efficient manipulation of nondeterministic TAs. 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. On the other hand, the semi-symbolic encoding is suitable for applications that make use of such alphabets.

At the present time, the main application of the structures and algorithms implemented in Vata for handling explicitly encoded TAs are the Forester tool described previously and the tools implementing TA-based decision procedures for separation logic: Spen (see Section 4.1) and Slide [IRV14]. The semi-symbolic encoding of TAs has been used in our decision procedure for WS1S in Section 4.2 (where we use unary TAs in the place of finite automata).

5.3.1 Experimental Results

In order to illustrate the level of optimisation that has been achieved in Vata and that can be exploited in its applications, we compared its performance against Timbuk for union and intersection of more than 3 000 pairs of TAs. On average, Vata was over 20 000 times faster on union and over 100 000 times faster on intersection. The comparison of the implemented inclusion checking algorithms can be found in Section 5.1.

6 Conclusions and Future Directions

In this section, we summarise the main contributions of the thesis and discuss possible further research directions.

6.1 A Summary of the Contributions

The main focus of this thesis was on improving the state of the art in shape analysis. This high-level goal was addressed by contributions in the following three areas. In the first area of forest automata-based shape analysis, we developed an extension of the analysis proposed in [HHR$^+$12] that allows it to run fully automatically, without user intervention. The extension is based on learning boxes, i.e. lower-level forest automata describing repeated substructures of the considered complex dynamic data structure, which needed to be provided by the user in the original analysis. The boxes are inferred automatically from the structure of the sets of heap graph that occur during the run of the analysis. Moreover, we extended the analysis even further by considering the relations between the data stored in the heap cells. We trace ordering relations between the data stored, which allows us to verify programs such as various sorting algorithms (bubblesort and insertsort over lists), programs with binary search trees, or programs with skip lists of two and three levels.

In the second area, which focused on the development of decision procedures for various logics, we proposed the following two procedures: First, we proposed a decision procedure for testing entailment in a fragment of separation logic that contains various flavours of lists that appear in practice. The decision procedure is based on decomposing the whole entailment query into several lower-level queries and deciding those by translating them into the TA membership problem. Second, we proposed a decision procedure for testing validity of WS1S formulae. The decision procedure is based on transforming the formula to be decided into the prenex normal form, constructing a finite automaton for the matrix of the formula, and, finally, processing the prefix of the formula using a technique that is a generalisation of the antichain principle from testing universality and language inclusion of finite automata.

In the third area focusing on finding new and improving existing techniques for manipulating nondeterministic TAs, we contributed by the following results. We developed a new technique for testing language inclusion that is based on a downward traversal through the automaton. We further augmented the basic technique with the use of antichains and simulations, and also proposed more advanced optimisations. According to our experiments, the technique performs often better than the so far used upward inclusion checking, which is based on upward determinisation of the automaton. Moreover, we also proposed a semi-symbolic encoding of nondeterministic TAs and developed algorithms for automata operations (including some more advanced like computing the maximum downward simulation relation on the states of the automaton, or checking language inclusion of a pair of automata) over this encoding. Our work in exploring efficient techniques for handling nondeterministic TAs culminated in the development of the Vata library, where these techniques are implemented, and which is, as far as we know, currently the most efficient library for manipulating nondeterministic TAs available.

6.2 Further Directions

There are many interesting directions of further work. In the area of automata-based shape analysis, an interesting direction is to consider a more general notion than the currently used formalism of forest automata. One option would be to remove the restriction that the boxes cannot be recursive. Such a change would increase the expressive power of forest automata, allowing them to express such data structures as trees with linked leaves or skip lists of an arbitrary height. On the other hand, the box folding and learning algorithms would need to be significantly re-designed. Another option would be to adopt a different model, based e.g. on the encoding of inductive higher-order predicates used in the decision procedure for separation logic of Iosif *et al* [IRV14]. In any case, we wish to extend the forest automata-based shape analysis with a counterexample-guided abstraction refinement (CEGAR) loop and use predicate language abstraction on the forest automata instead of the coarse finite height abstraction used now. We believe that the use of the more refined abstraction should allow us to verify some data structures that we currently cannot handle due to the abstraction used, such as red-black trees.

A further interesting future direction is the development of an approach that would allow verification of memory allocators (such as the `ptmalloc()` allocator used in the `glibc` library), which is a truly challenging task due to the complex overlaid shape of the used data structures. A more general representation would also be needed for the verification of some concurrent programs with dynamic memory, e.g. lock-free implementations of concurrent skip lists. In this setting, the invariant of the sequential skip list, which we are currently able to infer, is broken in this particular lock-free concurrent setting, and forest automata, as defined, cannot represent it. Nevertheless, we plan to apply the shape analysis to verification of concurrent programs, combining it e.g. with the ideas of Abdulla *et al* [AHH$^+$13].

Regarding our decision procedure for separation logic, in future, we wish to continue with extending its generality. In particular, we would like to weaken the limitations on the Boolean structure of the formulae, and, moreover, we would also like to explore whether it is possible to combine it with the decision procedure from [IRV14], which considers more general inductive definitions. For the decision procedure for WS1S, there are several possibilities. We wish to extend the decision procedure to WSkS for an arbitrary k by the use of TAs and, probably, an algorithm with a structure similar to the structure of the algorithms for downward language inclusion testing of nondeterministic TAs that were presented in the thesis. We also plan to generalise our notion of symbolic terms in the algorithm to reduce the number of states of the automaton for the matrix of a formula. We believe that our proposed decision procedure is only the start of a new research direction searching for techniques for efficiently deciding WSkS formulae, combining heuristics from both automata theory and formal logic.

Even though the methods for manipulating nondeterministic finite tree (and word) automata have seen a great advance in the recent years, as shown by a recent algorithm for testing equivalence and inclusion of nondeterministic finite word automata of Bonchi and Pous [BP13], there is still a space for improvement. We wish to generalise their algorithm to testing inclusion of nondeterministic TAs. We also wish to keep exploring yet other possibilities for reducing the state space in checking language inclusion of nondeterministic finite and tree automata. Furthermore, one of our future goals is to develop an efficient technique for reducing nondeterministic finite automata, both for words and trees, going beyond the capabilities of the techniques based on the simulation equivalence. In the area of symbolic representation of finite word and tree automata, we wish to explore different encodings, suitable for particular needs, such as for the use in the decision procedures of various logics (e.g. WSkS) or for the verification of hardware.

6.3 Publications Related to this Thesis

The results presented in this thesis were originally published in the following papers. The automated approach for learning boxes in the forest automata-based shape analysis, together with the refined technique for abstraction appeared in [HLR$^+$13]. The data extension of the technique was published as [AHJ$^+$13] (and later extended in [AHJ$^+$15]). The decision procedures for separation logic was published in [ELSV14], and the decision procedure for WS1S has been accepted to appear as [FHLV15]. Our algorithms for manipulating nondeterministic TAs were published in [HLŠV11] (the downward inclusion checking and the algorithms for the semi-symbolic representation), and the description of Vata appeared in [LŠV12].

Bibliography

[ABH$^+$08] Parosh Aziz Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. Computing simulations over tree automata: Efficient techniques for reducing tree automata. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 93–108. Springer, 2008.

[ACH$^+$10] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains (on checking language inclusion of NFAs). In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 158–174. Springer, 2010.

[AHH$^+$13] Parosh Aziz Abdulla, Frédéric Haziza, Lukáš Holík, Bengt Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures. In *Proc. of TACAS'13*, volume 7795 of *LNCS*, pages 324–338. Springer, 2013.

[AHJ$^+$13] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In

Proc. of ATVA'13, volume 8172 of *LNCS*, pages 224–239. Springer, 2013.

[AHJ+15] Parosh Aziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, Cong Quy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. *Acta Informatica*, 2015. Accepted for publication.

[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, 2011.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[BGP12] James Brotherston, Nikos Gorogiannis, and Rasmus Lerchedahl Petersen. A generic cyclic theorem prover. In *Proc. of APLAS'12*, volume 7705 of *LNCS*, pages 350–367. Springer, 2012.

[BHH+08] Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *Proc. of CIAA'08*, volume 5148 of *LNCS*, pages 57–67. Springer, 2008.

[BHRV06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*, pages 52–70. Springer, 2006.

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

[BP13] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proc. of POPL'13*, pages 457–468. ACM, 2013.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM, 1977.

[CDNQ12] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006–1036, 2012.

[CJH⁺09] Philipp Claves, Dorothea Jansen, Sezar Jarrous Holtrup, Martin Mohr, Anton Reis, Maria Schatz, and Irene Thesing. The LETHAL library, 2009. Available from `http://lethal.sourceforge.net/`.

[DPV13] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-precise verification of low-level list manipulation. In *Proc. of SAS'13*, volume 7935 of *LNCS*, pages 215–237. Springer, 2013.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x: new techniques for WS1S and WS2S. In *Proc. of CAV'98*, volume 1427 of *LNCS*, pages 516–520. Springer, 1998.

[ELSV14] Constantin Enea, Ondřej Lengál, Mihaela Sighireanu, and Tomáš Vojnar. Compositional entailment checking for a fragment of separation logic. In *Proc. of APLAS'14*, volume 8858 of *LNCS*, pages 314–333. Springer, 2014.

[FHLV15] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Nested antichains for WS1S. In *Proc. of TACAS'15*, volume 9035 of *LNCS*, pages 658–674. Springer, 2015.

[Gen03] Thomas Genet. Timbuk/Taml: A tree automata library, 2003. Available from `http://www.irisa.fr/lande/genet/timbuk`.

[HHR⁺12] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest automata for verification of heap manipulation. *Formal Methods in System Design*, 41(1):83–106, 2012.

[HLR⁺13] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Fully automated shape analysis based on forest automata. In *Proc. of CAV'13*, volume 8044 of *LNCS*, pages 740–755. Springer, 2013.

[HLŠV11] Lukáš Holík, Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. Efficient inclusion checking on explicit and semi-symbolic tree automata. In *Proc. of ATVA'11*, volume 6996 of *LNCS*, pages 243–258. Springer, 2011.

[HNR10] Jonathan Heinen, Thomas Noll, and Stefan Rieger. Juggrnaut: Graph grammar abstraction for unbounded heap structures. In *Proc. of TTSS'09*, volume 266 of *ENTCS*, pages 93–107. Elsevier, 2010.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27, 2005.

[IRŠ13] Radu Iosif, Adam Rogalewicz, and Jiří Šimáček. The tree width of separation logic with recursive definitions. In *Proc. of CADE'13*, volume 7898 of *LNCS*, pages 21–38. Springer, 2013.

[IRV14] Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. Deciding entailments in inductive separation logic with tree automata. In *Proc. of ATVA'14*, volume 8837 of *LNCS*, pages 201–218. Springer, 2014.

[JJSK97] Jakob L. Jensen, Michael E. Jørgensen, Michael I. Schwartzbach, and Nils Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *Proc. of PLDI'97*, pages 226–234. ACM, 1997.

[KMS02] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.

[LGQC14] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Proc. of CAV'14*, volume 8559 of *LNCS*, pages 52–68. Springer, 2014.

[LŠV12] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A library for efficient manipulation of non-deterministic tree automata. In *Proc. of TACAS'12*, volume 7214 of *LNCS*, pages 79–94. Springer, 2012.

[MPQ11] Parthasarathy Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. In *Proc. of POPL'11*, pages 611–622. ACM, 2011.

[MQ11] Parthasarathy Madhusudan and Xiaokang Qiu. Efficient decision procedures for heaps using STRAND. In *Proc. of SAS'11*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.

[MS01] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proc. of PLDI'01*, pages 221–231. ACM, 2001.

[MTLT10] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proc. of POPL'10*, pages 211–222. ACM, 2010.

[PR11] Juan Navarro Pérez and Andrey Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *Proc. of PLDI'11*, pages 556–566. ACM, 2011.

[PR13] Juan Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In *Proc. of APLAS'13*, volume 8301 of *LNCS*, pages 556–566. Springer, 2013.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55–74. IEEE, 2002.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.

[TH03] Akihiko Tozawa and Masami Hagiya. XML schema containment checking based on semi-implicit techniques. In *Proc. of CIAA'03*, volume 2759 of *LNCS*, pages 213–225. Springer, 2003.

[WDHR06] Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.

[WMK11] Thomas Wies, Marco Muñiz, and Viktor Kuncak. An efficient decision procedure for imperative tree data structures. In *Proc. of CADE'11*, volume 6803 of *LNCS*, pages 476–491. Springer, 2011.

[YLB$^+$08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Proc. of CAV'08*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.

[ZKR08] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *Proc. of PLDI'08*, pages 349–361. ACM, 2008.

Curriculum Vitae

Personal Data

Name:	Ondřej Lengál
Nationality:	Czech Republic
Date of birth:	February 19, 1986
E-mail:	`ilengal@fit.vutbr.cz`
Homepage:	`http://www.fit.vutbr.cz/~ilengal/`

Education

since 2010 Brno University of Technology, Faculty of Information Technology, studying in Ph.D. study programme Computer Science and Engineering

2008 – 2010 Brno University of Technology, Faculty of Information Technology, master's degree in Information Technology, master's thesis *An Efficient Finite Tree Automata Library*

2005 – 2008 Brno University of Technology, Faculty of Information Technology, bachelor's degree in Information Technology, bachelor thesis *Automatic Generation of Processing Elements for FPGA*

2001 – 2005 Střední průmyslová škola Zlín, four year high school, specialization in low voltage electronics

Accomplishments

9/2010 First place in Master's Thesis of the Year (Diplomová práce roku) competition, section Information Safety, System Development Control Standards, and Interdisciplinary Approaches (Informační bezpečnost, standardy řízení vývoje systémů a mezioborové přístupy)

6/2010 Dean's prize for master's thesis

5/2010 EEICT student competition, Brno, paper *An Efficient Finite Tree Automata Library*, first place in category *Intelligent Systems*, master section

10/2009 Prize of Zdena Rábová

11/2008 ACM Student Research Competition, Praha, paper *Automatic Generation of Processing Elements for FPGA*, final round

5/2008 EEICT student competition, Brno, paper *Automatic Generation of Processing Elements for FPGA*, first place in category *Graphics and Computer Systems*, bachelor section

5/2007 GE Foundation Scholar-Leaders scholarship

Work Experience

2008 – 2009	CESNET z.s.p.o., programmable hardware developer
2/2006 – 11/2006	Tribun s.r.o., developer and maintainer of information systems

Research Activities

since 2010 Member of VeriFIT, a research group at Faculty of Information Technology Brno University of Technology that focuses on formal verification. A particular focus on symbolic, automata-based methods of automatic verification of infinite-state systems, such as programs with complex dynamic linked data structures, automata-based decision procedures of logics, and techniques for efficient manipulation of nondeterministic finite state automata.

2006 – 2010 Research and development of hardware-accelerated network applications. Participation in the development of state-of-the-art network applications (such as filtering or monitoring) for multi-gigabit networks.

Language skills

Czech, English.

Abstract

The work presented in this thesis focuses on finite state automata over finite words and finite trees, and the use of such automata in formal verification of infinite-state systems. First, we focus on extensions of a previously introduced framework for verification of heap-manipulating programs—in particular programs with complex dynamic data structures—based on tree automata. We propose several extensions to the framework, such as making it fully automated or extending it to consider ordering over data values. Further, we also propose novel decision procedures for two logics that are often used in formal verification: separation logic and weak monadic second order logic of one successor (WS1S). These decision procedures are based on a translation of the problem into the domain of automata and subsequent manipulation in the target domain. Finally, we have also developed new approaches for efficient manipulation with tree automata, mainly for testing language inclusion and for handling automata with large alphabets and implemented them in a library for general use. The developed algorithms are used as the key technology to make the above mentioned techniques feasible in practice.