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

Ing. Tomáš Fiedor

Automata in Decision Procedures and Performance Analysis

Automaty v rozhodovacích procedurách a výkonnostní analýze

Extended Abstract of a Ph.D. Thesis

Supervisor: prof. Ing. Tomáš Vojnar, Ph.D.
Co-Supervisor: Doc. Mgr. Adam Rogalewicz, Ph.D.

Key Words

Antichains, amortized complexity, binary decision diagrams, finite automata, forest automata, formal verification, heap-manipulating programs, monadic logic, non-determinism, resource bounds analysis, second-order logic, shape analysis, shape norms, static analysis, tree automata, ws1s.

Klíčová slova

Analýza mezí zdrojů, analýza tvaru, antiřetězce, amortizovaná složitost, binární rozhodovací diagramy, formální analyza, konečné automaty, logika druhého řádu, lesní automaty, monadická logika, programy manipulující s haldou, nedeterminismus, statická analýza, stromové automaty, tvarové normy, ws1s.

Contents

1 Introduction

For almost a century, computer technology has been a necessary part of our lives: mankind exploits it in everyday life, in medicine, in transportation or in heavy industry. Naturally, computer programs should be error-free, since any error can have either little (such as bad user experience), medium (such as cash loss) or even severe consequences (such as accidents or crashes). To prevent these errors before-hand, one can try to use formal analysis and verification, however, these techniques still face a great challenge: complex systems leads to an analysis of infinite state space, and, in many cases, even to infeasibility. While, for many classes of programs we can prove or analyse many properties (including safety, termination or atomicity of programs), every year the list keeps growing with many new properties to be checked and many new characteristics of programs to be analysed leading to brand new challenges

For instance recently, developers have been more frequently demanding tools that would help them understand the performance of their code. In some cases, they even need to verify that their programs stay within the expected resource bounds (i.e. bounds on the expected consumption of computational time, memory, disk space, energy, etc.) or at least obtain a reasonable estimate of the program performance. In their software, *performance-related issues* are common and lead to a poor user experience or a waste of computational resources, as is documented by many recent studies [JSS+12]. These studies claim that the root cause of all of these issues is that developers do not understand the performance of their programs enough. But there are many other important factors involved in such widespread: insufficient performance regression testing, small test workloads or the fast development frequently breaking the codebase.

Unlike in the case of functional bugs, a large percentage of performance bugs is usually discovered through code reasoning or profiling, and not through the majority of users reporting negative effects of the bugs or through regular automated checks. Performance degradations are subtle, and they tend to manifest only with considerably big workloads. In the end they are missed by the frequent regression testing and noticed only by individual users in individual cases. So, naturally, techniques to help developers reason about the performance, better test oracles or better profiling techniques are needed in order to discover these kinds of bugs early in the process. Obviously, we have to extend the developer's everyday toolbox with efficient automated performance analyses and automated detection of performance bugs.

Although some research of automated performance analysis has already been done, the currently known techniques are still far from being satisfactory. This is especially true when the analysed code works not only with simple data

types such as integers, but employs complex dynamic data structures based on pointers such as lists, trees, and their various combinations or extensions.

Such data structures are commonly used in complex system code like operating system kernels, compilers, database engines, browsers and even embedded systems, whose poor performance can significantly impact the user experience. It is a well-known fact that dynamic data structures are hard to develop and can contain intricate errors which, in addition, only manifest under certain circumstances and are thus difficult to track down. Moreover, in performance critical applications, developers use even more advanced data structures such as, e.g. red-black trees, priority heaps, or lock-free linked lists, as well as various advanced programming techniques like, e.g. pointer arithmetic or block operations for performance speed up. For programs based on such techniques, even safety verification is still a challenge and works on their automated performance analysis are extremely rare.

In theory we can divide performance analysis into two main approaches — static and dynamic analysis. For an input program, the first one allows us to infer theoretically proven resource bounds; on the other hand, the latter collects performance records from one or more program runs, possibly extrapolates these data and then only estimates resource bounds, without any theoretical proof. But while both approaches have their advantages, e.g. in terms of the speed or precision, and the right time to be used, they share many challenges that must be overcome to apply them in everyday development.

One of these challenges is choosing a suitable formal theory to describe the program invariants. We need a theory that allows for a scalable analysis to be implemented on top of it and that is, at the same time, expressive enough to be able to reason about properties of advanced structures, especially their shapes or resource bounds. Commonly, researches use logics due to their great expressive power. However, with such power comes the price: great complexity of the associated decision problems, with some logics even being undecidable. In order to achieve an efficient analysis, one then has to improve the state of the art or use dedicated theories such as, e.g. separation logic [Rey02], or weak monadic second order logic with one successor (WS1S) [Büc59].

The latter, WS1S, has lots of applications not only for reasoning over the data structures [MPQ11, ZHW$^+$14]. It is still a decidable logic, however, its decision problem lies in the NONELEMENTARY class: it lurks on the borders of decidability. So while many WS1S formulae are decidable in a reasonable time, sometimes its complexity simply strikes back. And then we have to either fight back or give up building on the WS1S at all. But, we hope we could exploit the recent advancements in automata theory, e.g. the antichain principles, to push the usability border of WS1S even further.

Another challenge is how to build such analysers. In particular, in the area

of resource bounds analysis, current static resource bounds analyses are so far mostly limited to programs with integer variables only. When pointers are used in the analysed programs the analysers usually return a huge number of false negatives, not knowing the precise targets of the used pointers or the shape of the dynamic data structures being handled. They are forced to work with basic assumptions over the pointer variables, and thus they have to sacrifice soundness or precision of the approach. Programs with pointers are, however, common in practice, so this limitation is rather significant from the point of view of applicability.

Alternatively, we can give up the precision of the static analysis, focus on dynamically captured data and only settle for estimates of the program performance from concrete program runs. While one cannot guarantee how the program under analysis will perform or whether it will trigger any bug, dynamic analysis can still provide a useful insight and can be exploited, e.g. to detect performance changes or to infer statistical models of expected performance, leading to a better program understanding.

In the end, both static and dynamic analyses have their own shortcomings, nevertheless, we, researchers, should also focus our efforts on the developer experience. We should always strive to achieve a high performance bug fix ratio instead of high bug detection ratio since only that shows that our methods are applicable in practice. Every performance analysis should provide at least (i) approximate location where the bug was located, (ii) estimated severity how the performance or functionality is influenced by the bug, and (iii) detection confidence whether the bug was real or spurious. These factors greatly affect whether the developer will confirm the bug the subsequent decision whether it should be fixed. However, most importantly, if these bugs are to be fixed at all, developers have to catch them early in the development process when their mindset is still in the context of the influenced code. This can only be achieved by integrating static and dynamic analysers into the existing development workflows such as the continuous integration.

1.1 Goals of the Thesis

The aim of this thesis is to extend the current state of the art of formal analysis and verification of systems with infinite state space and with the focus on techniques based on automata. In particular, we address this goal in two distinct parts. On one hand, the thesis focuses on developing novel methods based on static analysis and, on the other hand, also on enhancing methods for deciding formal theories, that are currently used in existing methods, to enable analysis and verification of a broader range of programs.

The first goal is enhancing the current methods for deciding selected logics.

In particular, the focus is put on decision procedures for the weak monadic second order logic of one successor (WS1S), which is the target of the translation of, e.g. logics of STRAND [MPQ11] and UABE [ZHW+14] — logics allowing expression of invariants of advanced data structures and arrays respectively. The current state-of-the-art decision procedures, however, are not efficient enough to decide more complex formulae and so authors of STRAND and UABE had to find a workaround in order to apply them properly in the field of program verification. Motivated by this situation, one of the goals is thus to improve the current state of the art in WS1S decision procedures, and to make them more efficient to be usable on more complex formulae such as those of STRAND or UABE.

The other goal focuses on performance analysis for heap-manipulating programs (with the emphasis on resource bounds analysis and automatic complexity analysis). While the current state-of-the-art of the resource bounds analysis of integer programs is already quite advanced, the state of the art of performance analysis of heap-manipulating programs is much less developed. We build on results from the fields of shape analysis [HŠRV13] and resource bounds analysis of integer programs [SZV17] to develop a sound analysis for verification of resource bounds of programs manipulating with advanced data structures. The key to solving this goal lies in a proper definition of so-called shape norms — numerical measures on data structures, such as the length of list or the number of elements in trees. Hence, the goal is to propose a flexible and powerful class of norms that will allow one to analyse a wide selection of data structures, such as binary trees or even skip-lists.

2 Preliminaries

We first introduce the notion of *weak monadic second-order logic of one successor*. Moreover, we present a brief introduction to automata theory.

Syntax. WS1S is a monadic second-order logic over the universe of discourse $\mathbb{N}_0$. This means WS1S supports second-order *variables*, usually denoted using upper-case letters $X, Y, \ldots$, that range over finite subsets of $\mathbb{N}_0$, e.g. $X = \{0, 3, 42\}$. Given X and Y are variables, we can defined WS1S atomic formulae as follows: (i) $X \subseteq Y$, i.e. the standard set inclusion, (ii) $\mathrm{Sing}(X)$, i.e. the singleton predicate, (iii) $X = \{0\}$, i.e. X is a singleton containing 0, and (iv) $X = Y + 1$, i.e. $X = \{x\}$ and $Y = \{y\}$ are singletons and x is the successor of y, i.e. $x = y + 1$. More complex formulae can be built using the classical logical connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), and the quantifier $\exists\mathbb{X}$ (existential quantification) where $\mathbb{X}$ is a finite set of variables (we write $\exists X$ if $\mathbb{X}$ is a singleton $\{X\}$).

Semantics. A *model* of a WS1S formula $\varphi(\mathcal{X})$ with the set of free variables $\mathcal{X}$ is an assignment $\rho : \mathcal{X} \to 2^{\mathbb{N}_0}$ of the free variables $\mathcal{X}$ of φ to finite subsets of $\mathbb{N}_0$ for which the formula is *satisfied*, written $\rho \models \varphi$. Satisfaction of atomic formulae is defined as follows: (i) $\rho \models X \subseteq Y$ iff $\rho(X) \subseteq \rho(Y)$, (ii) $\rho \models \mathrm{Sing}(X)$ iff $\rho(X)$ is a singleton set, (iii) $\rho \models X = \{0\}$ iff $\rho(X) = \{0\}$, and (iv) $\rho \models X = Y + 1$ iff $\rho(X) = \{x\}, \rho(Y) = \{y\}$, and $x = y + 1$. Satisfaction of formulae formed using logical connectives is defined as usual. A formula φ is *valid*, written $\models \varphi$, iff all assignments of its free variables to finite subsets of $\mathbb{N}_0$ are its models, and *satisfiable* if it has a model. Otherwise it is *unsatisfiable*.

Finite Automata Let $\mathbb{X}$ be a set of variables. A *symbol* τ over $\mathbb{X}$ is a mapping of all variables in $\mathbb{X}$ to either 0 or 1, e.g. $\tau = \{X_1 \mapsto 0, X_2 \mapsto 1\}$ for $\mathbb{X} = \{X_1, X_2\}$. An *alphabet* over $\mathbb{X}$ is the set of all symbols over $\mathbb{X}$, denoted as $\Sigma_{\mathbb{X}}$. For any $\mathbb{X}$ (even empty) we use $\bar{0}$ to denote the symbol which maps all variables from $\mathbb{X}$ to 0, $\bar{0} \in \Sigma_{\mathbb{X}}$, the so-called *zero symbol*.

A (non-deterministic) *finite* (word) *automaton* (abbreviated as FA in the following) over a set of variables $\mathbb{X}$ and an alphabet $\Sigma_{\mathbb{X}}$ is a quadruple $\mathcal{A} = (Q, \delta, I, F)$ where Q is a finite set of states, $I \subseteq Q$ is a set of *initial* states, $F \subseteq Q$ is a set of *final* states, and $\delta \subseteq Q \times \Sigma_{\mathbb{X}} \times Q$ is a set of transitions of the form (p, τ, q) where $p, q \in Q$ and $\tau \in \Sigma_{\mathbb{X}}$. We use $p \xrightarrow{\tau} q \in \delta$ to denote that $(p, \tau, q) \in \delta$. Note that for an FA $\mathcal{A}$ over $\mathbb{X} = \emptyset$, $\mathcal{A}$ is a unary FA with the alphabet $\Sigma_{\mathbb{X}} = \{\bar{0}\}$.

A *run* r of $\mathcal{A}$ over a word $w = \tau_1 \tau_2 \ldots \tau_n \in \Sigma_{\mathbb{X}}^*$ from the state $p \in Q$ to the state $s \in Q$ is a sequence of states $r = q_0 q_1 \ldots q_n \in Q^+$ such that $q_0 = p$, $q_n = s$ and for all $1 \leq i \leq n$ there is a transition $q_{i-1} \xrightarrow{\tau_i} q_i$ in δ. If $s \in F$, we say that r is an *accepting run*. We write $p \overset{w}{\Longrightarrow} s$ to denote that there exists a run from the state p to the state s over the word w. The *language* accepted by a state q is defined by $\mathcal{L}_{\mathcal{A}}(q) = \{w \mid q \overset{w}{\Longrightarrow} q_f, q_f \in F\}$; the language of a set of states $S \subseteq Q$ is defined as $\mathcal{L}_{\mathcal{A}}(S) = \bigcup_{q \in S} \mathcal{L}_{\mathcal{A}}(q)$. When it is clear which FA $\mathcal{A}$ we refer to, we only write $\mathcal{L}(q)$ or $\mathcal{L}(S)$. The language of $\mathcal{A}$ is then defined as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_{\mathcal{A}}(I)$. We say that the state q accepts w and that the automaton $\mathcal{A}$ accepts w to express that $w \in \mathcal{L}_{\mathcal{A}}(q)$ and $w \in \mathcal{L}(\mathcal{A})$ respectively. We call a language $\mathcal{L} \subseteq \Sigma_{\mathbb{X}}^*$ *universal* iff $\mathcal{L} = \Sigma_{\mathbb{X}}^*$. For a set of states $S \subseteq Q$, we define $post_{[\delta,\tau]}(S) = \bigcup_{s \in S} \{t \mid s \xrightarrow{\tau} t \in \delta\}$.

We define the *complement* of an automaton $\mathcal{A}$ as the automaton $\mathcal{A}_{\mathcal{C}} = (2^Q, \delta_{\mathcal{C}}, \{I\}, \downarrow\{Q \setminus F\})$ where $\delta_{\mathcal{C}} = \left\{ P \xrightarrow{\tau} post_{[\delta,\tau]}(P) \;\middle|\; P \subseteq Q \right\}$, and $\downarrow\{Q \setminus F\}$ is the set of all subsets of Q that do not contain any final state of $\mathcal{A}$; this corresponds to the standard procedure that first determinizes $\mathcal{A}$ by the subset construction and then swaps its sets of final and non-final states. The language of $\mathcal{A}_{\mathcal{C}}$ is the complement of the language of $\mathcal{A}$, i.e. $\mathcal{L}(\mathcal{A}_{\mathcal{C}}) = \overline{\mathcal{L}(\mathcal{A})}$.

For a set of variables $\mathbb{X}$ and a variable X, the *projection* of X from $\mathbb{X}$,

denoted as $\pi_X(\mathbb{X})$, is the set $\mathbb{X} \setminus \{X\}$. For a symbol τ, the projection of X from τ, denoted $\pi_X(\tau)$, is obtained from τ by restricting τ to the domain $\pi_X(\mathbb{X})$[1]. For a transition relation δ, the projection of X from δ, denoted as $\pi_X(\delta)$, is the transition relation $\left\{p \xrightarrow{\pi_X(\tau)} q \mid p \xrightarrow{\tau} q \in \delta\right\}$.

A *word* over a finite alphabet Σ is a finite sequence $w = a_1 \cdots a_n$, for $n \geq 0$, of symbols from Σ. Its i-th symbol a_i is denoted by $w[i]$. For $n = 0$, the word is the empty word ϵ. A language L is a set of words over Σ. We use the standard language operators of concatenation $L.L'$ and iteration L^*. The (right) quotient of a language L w.r.t the language L' is the language $L - L' = \{u \mid \exists v \in L' : uv \in L\}$. We abuse the notation and write $L - w$ to denote $L - \{w\}$, for a word $w \in \Sigma^*$.

3 Decision Procedures for WS1S

Weak monadic second-order logic of one successor (WS1S) is a powerful language for reasoning about regular properties of finite words. It has, indeed, found numerous applications, ranging from software and hardware verification through controller synthesis to, e.g. computational linguistics or verification of parametric systems. Most of these successful applications were possible due to the well-known MONA tool [EKM98], which implements classical automata-based decision procedures for WS1S and WS2S logics (a generalization of WS1S to finite binary trees). However, the worst-case complexity of WS1S is NONELEMENTARY [Mey72], and, despite many optimizations implemented in MONA and other tools, the complexity sometimes simply strikes back. However, for a logic as expressive as WS1S any further advancements in its decision procedures could improve its practical applicability as well as open new applications, e.g. for performance or resource bounds analysis.

3.1 Nested Antichains for WS1S

The classical approach for deciding WS1S, e.g. as implemented within the MONA tool, works with deterministic automata. It uses determinization extensively, and it relies on efficient minimization of deterministic automata to suppress the complexity blow-up. However, the worst-case exponential complexity of determinization often significantly harms the performance of the tool. But we believe that we can alleviate this problem by exploiting some of the recent works on efficient methods for handling non-deterministic automata—in par-

[1]Note there are several ways how to restrict the symbol to the domain — either by removing the track corresponding to the variable from the transitions or pump the transition relation by so-called *don't cares*, i.e. the track will contain both 0 or 1.

ticular, works on efficient testing of language inclusion and universality of finite automata [ACH+10] and works on reducing the size of finite automata using simulation relations [ABH+08]. These methods can handle non-deterministic automata while avoiding the determinization, and it has been shown they provide great efficiency improvements[2], e.g. in shape analysis. We thus make a major step towards building the entire decision procedure of WS1S on non-deterministic automata using similar techniques. We propose a generalization of the antichain algorithms of [DR10] to address the main bottleneck of the automata-based decision procedure for WS1S, i.e. the source of its complexity: the elimination of alternating quantifiers, which — when implemented on the automata level — produces nondeterministic FAs, and is followed by determinisation needed to allow subsequent negations.

The classical automata-based decision procedure translates the input WS1S formula into a finite word automaton such that its language represents all models of the formula. The automaton is built in a bottom-up manner according to the syntactic structure of the formula, starting with predefined automata for its literals (called "atomic" automata in the following) and applying a corresponding automata operation for every logical connective and quantifier ($\wedge, \vee, \neg, \exists$).

We can explain the source of the nonelementary complexity of the procedure on an example formula of the form $\varphi' = \exists X_m \forall X_{m-1} \ldots \forall X_2 \exists X_1 : \varphi_0$. First, we replace universal quantifiers by negation and existential quantification, which results into the formula $\varphi = \exists X_m \neg \exists X_{m-1} \ldots \neg \exists X_2 \neg \exists X_1 : \varphi_0$. The algorithm then builds a sequence of automata for the sub-formulae $\varphi_0, \varphi_0^\sharp, \ldots, \varphi_{m-1}, \varphi_{m-1}^\sharp$ of φ where $\varphi_i^\sharp = \exists X_{i+1} : \varphi_i$ and $\varphi_{i+1} = \neg \varphi_i^\sharp$ for $0 \leq i < m$. Every automaton in the sequence is constructed from the previous one by applying automata operations corresponding to negation or elimination of the existential quantifier. The latter corresponds to modifying automata transitions and may potentially introduce non-determinism.

However, the typical approach of complementing an NFA, i.e., determinising it first and then switching final and nonfinal states, may result into an exponential blowup: given an automaton for ψ, the automaton for $\neg\psi$ is constructed by the classical automata-theoretic construction consisting of determinization by the subset construction followed by swapping of the sets of final and non-final states. Since the subset construction is exponential in the worst case, the worst-case complexity of the procedure on the given φ is then a tower of exponentials with one level for every quantifier alternation in φ. Note that this high computational cost cannot be avoided completely—indeed, the nonelementary complexity is an inherent property of the problem.

[2]Naturally, the worst-case exponential complexity of these methods is an inherent property, however, the average complexity can indeed be improved so the methods can be used in practice.

An overview of the proposed algorithm. Instead we propose an algorithm for processing alternating quantifiers in the prefix of a formula which avoids the explicit determinization (and hence the associated exponential blow-up) of automata in the classical procedure and significantly reduces the state space explosion associated with it. Our algorithm is based on a generalization of the antichain principle used for deciding universality and language inclusion of finite automata [ACH+10]. We generalized the antichain algorithms so that instead of processing only one level of the chain of automata, we process the whole chain of quantifications with i alternations on-the-fly. Basically this means we are working with automata states that are sets of sets of sets ... of states of the automaton representing φ_0 of the nesting depth i (this corresponds to i levels of subset construction being done on-the-fly). In our algorithm we use nested symbolic terms to represent sets of such automata states and a generalized version of antichain pruning based on a notion of subsumption that descends recursively down the structure of the terms while pruning on all their levels.

However, note that our proposed nested antichain approach has its own limitations: currently we can only process a quantifier prefix of a formula, after which we return the answer to the validity query, but not an automaton representing all models of the input formula. That is, we cannot use the optimized algorithm for processing inner negations and alternating quantifiers which are not a part of the quantifier prefix or unground formulae.

Contributions. We summarize our contributions to WS1S achieved by our first proposed approach linked with the dWiNA tool:

1. By generalization of antichain techniques, we develop a decision procedure that can efficiently process long chains of quantifiers in the given formulae.

2. We show in our experimental evaluation that we improve the state of the art of WS1S decision procedures. In particular, we report on a series of parametric families of formulae, where we outperformed the state-of-the-art approaches.

3.1.1 Experimental Evaluation

We have implemented a prototype of our first approach in the tool called dWiNA [FHLV14]. We built it over the frontend of the Mona tool to parse the input formula into an internal representation in the form of FAs encoded using the MTBDD-based representation from the `libvata` library [LŠV12].

We evaluated dWiNA against several parametric families of manually constructed formulae, from which some were originally designed as show cases for evaluation of other tools. We compared dWiNA with the Mona tool, an implementation of the coalgebraic decision procedure [Tra15], which we refer to as Coalg, a decision procedure based on symbolic automata [DV], which we refer to as SFA, and the tool Toss implementing a procedure based on the Shelah's decomposition [GK10].

Since the tools support a limited set of syntactic features, we could only use a subset of the available benchmark formulae. Namely, we took the parametric families of formulae `HornLeq` from [DV] and `HornIn` from [GK10], originally proposed to evaluate the performance of SFA and Toss, respectively, and our parametric family of formulae `SetClosed`.[3] This experiment was run on a machine with a system that meets the requirements of all the tools[4], with an Intel Core i7-4770@3.4 GHz processor and 16GiB RAM, running Debian GNU/Linux. Table 1 gives the run times of the tools. We use ∞ in case the time exceeded 2 minutes and `oom` to denote that the tool ran out of memory. We can see that dWiNA outperforms the other tools on the most of the formulae.

Table 1: Experiments with parametric families of formulae

Benchmark	Mona	Toss	Coalg	SFA	dWiNA	**Benchmark**	Mona	Toss	Coalg	SFA	dWiNA
`HornLeq` [DV]						`SetClosed`					
horn-leq06	0.01	0.02	1.10	0.01	0.01	set-closed01	0.01	0.02	0.04	0.01	0.01
horn-leq07	0.01	0.02	11.09	0.01	0.01	set-closed02	0.01	0.02	∞	0.13	0.01
horn-leq08	0.01	0.02	101.48	0.01	0.01	set-closed03	0.01	0.18	∞	0.14	0.01
horn-leq09	0.01	0.02	∞	0.01	0.01	set-closed04	0.34	∞	∞	13.96	0.01
horn-leq11	0.05	0.03	∞	0.02	0.01	set-closed05	∞	∞	∞	∞	0.01
horn-leq13	0.19	0.04	∞	0.02	0.01	set-closed07	∞	∞	∞	∞	0.01
horn-leq14	0.45	0.04	∞	0.02	0.01	set-closed09	∞	∞	∞	∞	0.10
horn-leq15	1.19	0.05	∞	0.03	0.02	set-closed11	∞	∞	∞	∞	0.95
horn-leq16	3.35	0.05	∞	0.03	0.02	set-closed12	∞	∞	∞	∞	3.61
horn-leq17	9.07	0.05	∞	0.03	0.02	set-closed13	∞	∞	∞	∞	14.3
horn-leq18	22.89	0.06	∞	0.03	0.02	set-closed14	∞	∞	∞	∞	69.08
horn-leq19	`oom`	0.06	∞	0.03	0.03	set-closed15	∞	∞	∞	∞	∞

3.2 Lazy Automata Techniques for WS1S

We have said that the classical WS1S decision procedure builds an automaton A_φ accepting all encodings of models of the given formula φ in a form of finite words, and only then tests whether the language of A_φ is empty. The bottleneck of this procedure is the size of A_φ, which can be huge due to the fact that the derivation of A_φ involves many nested automata product constructions and complementation steps, preceded by determinization. We have demonstrated how one can avoid this bottleneck when processing the topmost quantifier prefix of the given formulae in our previosu method. However, we

[3]Note that the `HornSub` family is not supported by Toss and Coalg, and thus we chose a comparably complex family of `SetClosed` to present the overall comparison.

[4]Note that the Toss tool required specific version of OCaml that was not available for a stable debian build.

limited ourselves to processing of this prefix only and hence could not process arbitrary formula efficiently, and, moreover, in quite some cases the decision procedure still led to a state-space explosion.

Hence, our next goal is to avoid more of the state-space explosion involved in the *explicit* construction and to handle formulae without a need to transform them into the prenex normal form. We represent automata *symbolically* and, while constructing A_φ, we test the emptiness of A_φ *on the fly* which allows us to omit the state space irrelevant to the emptiness test. We build on two main principles: *lazy evaluation* and *subsumption-based pruning*. These principles have, to some degree, already appeared in the so-called antichain-based testing of language universality and inclusion of finite automata [WDHR06]. However, the richer structure of the WS1S decision problem allows us to elaborate on these principles in novel ways and utilize their power even more.

Overview of our algorithm. We propose an algorithm which originates in the classical WS1S decision procedure, in which models of formulae are encoded by finite words over a multi-track binary alphabet where each track corresponds to a variable of φ. In t_φ, the atomic formulae of φ are replaced by predefined automata accepting languages of their models. Boolean operators ($\wedge$, $\vee$, and $\neg$) are turned into the corresponding set operators ($\cup$, $\cap$, and complement) over the languages of models. An existential quantification $\exists X$ becomes a sequence of two operations. First, a projection π_X removes information about valuations of the quantified variable X from symbols of the alphabet, i.e. the so-called *projection* operation. After the projection, the resulting language L may, however, encode some but not necessarily *all* encodings of the models. In particular, encodings with some specific numbers of trailing $\bar{0}$'s, used as a padding, may be missing. To obtain a language containing *all* encodings of the models, L must be extended to include encodings with any number of trailing $\bar{0}$'s.

Here, we take the (right) $\bar{0}^*$-quotient of L, written $L - \bar{0}^*$, which is the set of all prefixes of words of L with the remaining suffix in $\bar{0}^*$. We give an example WS1S formula φ in equation (1) and its language term $t_{[\varphi]}$ in equation (2). The dotted operators represent operators over language terms.

$$\varphi \equiv \exists X : \mathrm{Sing}(X) \wedge (\exists Y : Y = X + 1) \tag{1}$$

$$t_{[\varphi]} \equiv \pi_X(\{\mathcal{A}_{\mathrm{Sing}(X)} \mathbin{\dot{\cap}} (\pi_Y(\mathcal{A}_{Y=X+1}) \mathbin{\dot{-}} \bar{0}^*)\}) \mathbin{\dot{-}} \bar{0}^* \tag{2}$$

The novelty of our work is that we test the emptiness of L_φ directly over $t_{[\varphi]}$. The term is used as a symbolic representation of the automata that would be explicitly constructed in the classical procedure: inductively to the terms structure, starting from the leaves and combining the automata of sub-terms by standard automata constructions that implement the term operators. Instead

of first building the automaton and only then testing emptiness, we test it on the fly during the construction. This offers opportunities to prune out portions of the state space that turn out not to be relevant for the test.

A sub-term $t_{[\psi]}$ of $t_{[\varphi]}$, corresponding to a sub-formula ψ, represents final states of the automaton $\mathcal{A}_\psi$ accepting the language encoding models of ψ. Predecessors of the final states represented by $t_{[\psi]}$ correspond to quotients of $t_{[\psi]}$. All states of $\mathcal{A}_\psi$ could hence be constructed by applying quotient operation on $t_{[\psi]}$ until fixpoint. By working with terms, our procedure can often avoid building large parts of the automata when they are not necessary for answering the emptiness query. For instance, when testing the emptiness of the language of a term $t_1 \cup t_2$, we adopt the *lazy approach* (in this particular case, the so-called *short-circuit evaluation*) and first test the emptiness of the language of t_1; if it is non-empty, we do not need to process t_2. Testing language emptiness of terms arising from quantified sub-formulae is more complicated since they translate to $-\bar{0}^*$ quotients. We evaluate the test on $t - \bar{0}^*$ by iterating the $-\bar{0}$ quotient from t. We either conclude with the positive result as soon as one of the iteration computes a term with a non-empty language, or with the negative one if the fixpoint of the quotient construction is reached. The fixpoint condition is that the so-far computed quotients *subsume* the newly constructed ones, where subsumption is a relation under-approximating inclusion of languages represented by terms. We also use subsumption to prune the set of computed terms so that only an *antichain* of the terms maximal wrt subsumption is kept.

Besides lazy evaluation and subsumption, our approach can benefit from multiple further optimizations. For example, it can be *combined* with the *explicit WS1S decision procedure*, which can be used to transform arbitrary sub-terms of t_φ to automata. These automata can then be rather small due to minimization, which cannot be applied in the on-the-fly approach (the automata can, however, also explode due to determinisation and product construction, hence this technique comes with a trade-off). We also propose a novel way of *utilising BDD-based encoding* of automata transition functions in the MONA style for computing quotients of terms. Finally, our method can exploit various methods of *logic-based pre-processing*, such as *anti-prenexing*, which, in our experience, can often significantly reduce the search space of fixpoint computations.

Contributions. We summarize our contributions to WS1S achieved by our second proposed approach linked with the Gaston tool:

1. Instead of the explicit automata construction, we develop an on-the-fly decision procedure based on the so-called language terms that can efficiently process arbitrary formulae. Contrary to the classical procedure,

our method can avoid costly determinisation in many cases.

2. We propose a combination of our procedure with the classical decision procedure for WS1S as implemented, e.g. by the MONA tool. This allows one to exploit the key optimizations of both approaches, i.e. minimization and lazy evaluation.

3. Besides the novel decision procedure, we develop a series of optimizations that are not limited to our approach only. Other tools and methods can exploit these optimizations, such as, e.g. anti-prenexing, to achieve better efficiency.

4. In our experimental evaluation, we demonstrate we improve the state of the art of WS1S decision procedures, especially on formulae describing program invariants of advanced data structures. In particular, we report on a series of benchmarks used for verification of programs, where we outperformed the MONA tool.

5. We perform an extensive evaluation of all the publicly available tools on a series of benchmarks that are used to stress-test WS1S decision procedures. We present a fair speed comparison of all of the available tools.

3.2.1 Experimental Evaluation

We have implemented the optimized procedure in a prototype tool GASTON [FHJ+16] [5]. Our tool uses the front-end of MONA to parse input formulae, to construct their corresponding abstract syntax trees, and to explicitly construct automata for sub-formulae.

We have compared GASTON's performance with that of MONA, our previous approach [FHLV15] implemented in the DWINA tool, the TOSS tool implementing the method of [GK10], and the implementations of the decision procedures of [Tra15] and [DV] (which we denote as COALG and SFA, respectively). In our experiments, we consider formulae obtained from various formal verification tasks as well as parametric families of formulae designed to stress-test WS1S decision procedures. We performed the experiments on a machine with the Intel Core i7-2600@3.4 GHz processor and 16 GiB RAM running Debian GNU/Linux.

[5] The name was chosen to pay homage to Gaston, an Africa-born brown fur seal who escaped the Prague Zoo during the floods in 2002 and made a heroic journey for freedom of over 300 km to Dresden. There he was caught and subsequently died due to exhaustion and infection.

Table 2: The comparison of MONA and GASTON on UABE benchmark.

Formula	MONA		GASTON	
	Time	**Space**	**Time**	**Space**
a-a	**1.71**	30 253	∞	∞
ex10	**7.71**	131 835	12.67	82 236
ex11	4.40	2 393	**0.18**	4 156
ex12	**0.13**	2 591	6.31	68 159
ex13	**0.04**	2 601	1.19	16 883
ex16	**0.04**	3 384	0.28	3 960
ex17	3.52	165 173	**0.17**	3 952
ex18	**0.27**	19 463	∞	∞
ex2	0.18	26 565	**0.01**	1 841
ex20	1.46	1 077	**0.27**	12 266
ex21	**1.68**	30 253	∞	∞
ex4	**0.08**	6 797	0.50	22 442
ex6	**4.05**	27 903	22.69	132 848
ex7	0.90	857	**0.01**	594
ex8	7.69	106 555	**0.03**	1 624
ex9	7.16	586 447	9.41	412 417
fib	**0.10**	8 128	24.19	126 688

A comparison of Gaston with Mona on UABE formulae. In Table 2, we show results of our experiments with formulae from the recent work of [ZHW+14] (denoted as UABE below), which uses WS1S to reason about programs with unbounded arrays of bounded elements . For this set of experiments, we considered MONA and GASTON only since the other tools were missing key features (e.g. atomic predicates) needed to handle the formulae.

The tables compare the overall time (in seconds) the tools needed to decide the formulae, and they also try to characterize the sizes of the generated state spaces. For the latter, we count the overall number of states of the generated automata for MONA, and the overall number of generated sub-terms for GASTON. The tables contain just a part of the results, the full results can be found in [FHJ+16]. We use ∞ in case the running time exceeded 2 minutes, oom to denote that the tool ran out of memory, $+k$ to denote that we added k quantifier alternations to the original benchmark, and N/A to denote that the benchmark requires some key feature or atomic predicate unsupported by the given tool. The results thus confirm that our approach can defeat MONA on many formulae in practice.

A comparison of Gaston with other tools. The second part of our experiments concerns parametric families of WS1S formulae used for evaluation in [GK10, FHLV15, DV], and also parametric versions of selected UABE formulae [ZHW+14]. Each of these families has one parameter (whose meaning is explained in the respective works). Table 3 gives times needed to decide instances of the formulae for the parameter having value of 20. If the tools

did not manage this value of the parameter, we give in parentheses the highest value of the parameter for which the tools succeeded. In this set of experiments, GASTON managed to win over the other tools on many of their own benchmark formulae.

Table 3: Experiments with parametric families of formulae

Benchmark	MONA	DWINA	TOSS	COALG	SFA	GASTON
`HornLeq` [DV]	`oom`(18)	**0.03**	0.08	∞(08)	**0.03**	**0.01**
`HornLeq` (+3) [DV]	`oom`(18)	∞(11)	0.16	∞(07)	∞(11)	**0.01**
`HornLeq` (+4) [DV]	`oom`(18)	∞(13)	**0.04**	∞(06)	∞(11)	**0.01**
`HornIn`[GK10]	`oom`(15)	∞(11)	0.07	∞(08)	∞(08)	**0.01**
`HornTrans` [FHLV15]	86.43	∞(14)	`N/A`	`N/A`	38.56	**1.06**
`SetSingle` [FHLV15]	`oom`(04)	∞(08)	0.10	`N/A`	∞(03)	**0.01**
`Ex8` [ZHW+14]	`oom`(08)	`N/A`	`N/A`	`N/A`	`N/A`	**0.15**
`Ex11(10)` [ZHW+14]	`oom`(14)	`N/A`	`N/A`	`N/A`	`N/A`	**1.62**

4 Using Static Analysis for Performance Analysis

The state of the art of static performance analysis of C and C++ programs focuses mostly on resource bounds and termination analysis of integer programs. While this field is currently well-established, works focusing on heap-manipulating programs are rather rare since they require precise analysis of the shape of the heap. Moreover, most of the works on shape analysis are usually limited to linear structures or have a hard-coded support for a single data structure.

Researchers usually transform the input heap-manipulating program into an integer one that is equal from the point of view of its termination or complexity. However, one has to first define the so-called shape numerical measures (or shape norms), that simulate the original data structures in the universe of integer programs. An example of a shape norm can be, e.g. the number of nodes in a tree, the length of a singly-linked list, or the height of a tree. Defining a suitable class of shape norms and inferring how their values change upon execution of program statements is the biggest challenge of such a transformation. Some classes are restricted to concrete data structures, such as singly-linked lists or trees. Other are applicable for a limited range of program constructions only, and cannot show resource bounds in many cases, e.g. when the resource bounds depend both on the shape of the heap and some integer constraints. Finally, many classes are not fully automatically usable and require manual involvement of the user.

4.1 From Shapes to Amortized Complexity

We first define a new parametric class of shape norms expressing the distance between two distinct points (memory cells pointed by some program variables and/or selector chains (so-called access paths); we can refer to these as "pointers") in the shape over some selector paths (such as `left`, `right` or `next` field), which conforms to a (restricted) regular expression. Moreover, we show how one can automatically derive a set of shape norms from the control flow graph (CFG) of a program. We strive to get a small set of the norms such that the analysis is as efficient as possible. If this set turns out not to be sufficient, it can later be (automatically) refined. This way the approach becomes fully automated. Based on this new class of shape norms, we then propose an approach to resource bounds analysis that exploits state-of-the-art shape analysers to transforms the input heap-manipulating program to a corresponding integer program. The resulting integer program can then be analysed similarly as in other existing works in the area. In our experiments we show, we improve on earlier results along several dimensions aiming at the automated resource bounds analysis of heap-manipulating programs that cannot be handled by existing approaches.

Overview of our approach. Our analysis works in three major steps. We first run a shape analysis and annotate the program with shape invariants. Second, using the results from the shape analysis, we create a corresponding integer abstraction of the program based on numeric information about the heap. Finally, we perform resource bound analysis purely on the resulting integer program.

The integer abstraction is based on our new class of *shape norms*, i.e. numerical measures on dynamic data structures (e.g. the length of a linked list). Our first contribution in this chapter is the definition of a class of shape norms that expresses the longest distance between two points of interest in a shape graph defined in terms of basic concepts from graph theory. We propose a class of norms that are parametric by the program under analysis and that are extracted in a pre-analysis (with a possibility of extending the initial set of tracked norms during the subsequent analysis); the extracted norms then correspond to the selector paths found in the program.

The second contribution is a calculus for our class of shape norms that allows us to derive how the norms change along a program statement, i.e. if the norm is incremented, resp. decremented, or reset to some other expression. The calculus consists of two kinds of rules. (1) Rules that allow one to directly infer the change of a norm and do not need to take any additional information into account. (2) Rules that rely on the preceding shape analysis; the shape information is mainly used there for (a) dealing with pointer aliasing and (b)

deriving an upper bound on the value of a norm from the result of the shape analysis (if possible). We point out that rules of the second kind encapsulate the points of the analysis where information about the shape is needed, and thus describe the minimal requirements on the preceding shape analysis. We believe that this separation of concern also allows one to use various other shape analysers if they satisfy the given criteria.

When creating the integer abstraction of the given heap-manipulating program, we could use all shape norms that we extracted from the program. However, we have an additional pre-analysis phase that eliminates norms that are not likely to be useful for the later bounds analysis. This reduction of norms has the benefit that it keeps the number of variables in the integer abstraction small. The number of extracted norms can be quadratic in the size of the program in the worst case, and adding quadratically many variables can be prohibitively expensive. The pre-analysis is therefore crucial to the efficiency of the later bound analysis. Moreover, the smaller number of additional variables increases the readability of the resulting integer abstraction and simplifies the development and debugging of subsequent analyses.

Finally, we perform resource bound analysis on the created integer abstraction. This design decision has two advantages. First, we can leverage the existing research on resource bound analysis for integer programs and do not have to develop a new bound analysis at all. Second, being able to analyse not only the shape but also integer changes has the advantage that we can analyse programs which mix integer iterations with data structure iterations; we illustrate this point on the flagship example of [Atk11], which combines iteration over data structures and integer loops in an intricate way. So far, no other approach has inferred precise resource bounds for this loop.

Contributions. We summarize our contributions to resource bounds analysis of heap-manipulating programs:

1. In comparison with related approaches, we consider a wider class of shape norms.

2. We develop a calculus for deriving the numeric changes of the considered shape norms. The rules of our calculus precisely identify the information that is needed from a shape analyser. We believe that this definition of minimal shape information is useful for development of future resource bound analysis tools for programs with recursive data structures as well.

3. Our norms are not fixed in advance but derived from the program to be verified: we define a pre-analysis that reduces the number of considered

Table 4: Experimental results.

Benchmark	Short description	Real bounds	Ranger				AProVE		Costa	
			Bound	SA	IG	BA	Time	Bound	Time	Bound
Basic										
SLL-cst	Constant-length SLL Traversal	$\mathcal{O}(1)$	$\mathcal{O}(1)$	0.002s	0.023s	0.011s	$\mathcal{O}(1)$	3.664s	$\mathcal{O}(n)$	0.251s
SLL	SLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.012s	0.087s	0.040s	$\mathcal{O}(n)$	6.434s	$\mathcal{O}(n)$	0.441s
SLL-nested	SLL with non-reset nested traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.027s	0.256s	0.057s	$\mathcal{O}(n)$	6.361s	$\mathcal{O}(n^2)$	1.582s
SLL-int	SLL Traversal with int combination	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.037s	0.275s	0.057s	$\mathcal{O}(n)$	8.945s	$\mathcal{O}(n)$	0.921s
CSLL	CSLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.013s	0.086s	0.032s	ERROR		Unknown	0.383s
CSLL-nt	Non-terminating CSLL Traversal	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.003s	0.001s	0.011s	ERROR		Unknown	0.843s
Advanced Structures										
DLL-next	Forward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.034s	0.518s	0.036s	$\mathcal{O}(n)$	5.954s	Unknown	0.657s
DLL-prev	Backward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.031s	0.181s	0.044s	$\mathcal{O}(n)$	6.459s	Unknown	0.712s
DLL-nt	Non-terminating DLL Traversal	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.011s	0.004s	0.024s	ERROR		Unknown	0.684s
DLL-int	Forward DLL Traversal with int combination	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.044s	0.654s	0.044s	$\mathcal{O}(n)$	5.723s	Unknown	0.946s
DLL-par	Parallel Forward and Backward DLL Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.058s	0.510s	0.069s	ERROR		Unknown	0.668s
Butterfly	Terminating Butterfly Loop	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.005s	0.054s	0.024s	$\mathcal{O}(n)$	7.389s	$\mathcal{O}(n)$	0.883s
Butterfly-int	Terminating Butterfly Loop with int combination	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	0.026s	0.198s	0.059s	$\mathcal{O}(n)^*$	3.513s	Unknown	0.899s
Butterfly-nt	Non-terminating Butterfly Loop	$\mathcal{O}(\infty)$	$\mathcal{O}(\infty)$	0.005s	0.090s	0.015s	$\mathcal{O}(n)^*$	7.768s	Unknown	1.701s
BST-double	Leftmost BST Traversal with nested Rightmost	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	25.147s	12.523s	0.203s	$\mathcal{O}(n^2)^{**}$	14.547s	Unknown	3.004s
BST-left	Leftmost BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.947s	7.321s	0.171s	$\mathcal{O}(n)^{**}$	13.335s	Unknown	2.476s
BST-right	Rightmost BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	2.895s	5.779s	0.168s	$\mathcal{O}(n)^{**}$	13.007s	Unknown	2.457s
BST-lr	Random BST Traversal	$\mathcal{O}(n)$	$\mathcal{O}(n)$	3.331s	7.010s	0.188s	$\mathcal{O}(n)^{**}$	14.488s	Unknown	2.619s
2-lvl SL-l1	2-lvl Skip-list Traversal via lvl1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.309s	0.837s	0.036s	ERROR		Unknown	1.449s
2-lvl SL-l2	2-lvl Skip-list Traversal via lvl2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.096s	0.526s	0.042s	ERROR		Unknown	1.442s
Advanced Algorithms										
FuncQueue	Queue implemented by two SLLs	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.046s	0.519s	0.136s	$\mathcal{O}(n)$	8.222s	Unknown	4.808s
Partitions	SLL Partitioning	$\mathcal{O}(n)$	$\mathcal{O}(n)$	0.094s	0.729s	0.059s	$\mathcal{O}(n^2)$	8.526s	$\mathcal{O}(n^2)$	7.047s
InsertSort	Insert Sort on SLL	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	0.041s	0.288s	0.051s	$\mathcal{O}(n^2)$	6.453s	$\mathcal{O}(n^2)$	0.904s
MergeInner	Showcase example of Atkey [Atk11]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	3.589s	14.080s	1.502s	$\mathcal{O}(n^2)$	57.935s	TIMEOUT(60s)	

norms. To our experience, this reduction is especially affecting the efficiency of the underlying resource bounds analysers, but it is also useful for reporting the derived integer abstraction to the user.

4. We demonstrate in an experimental validation that we obtain a powerful analysis. We report on iterations over complex data structures that could not be analysed before and discuss our fully-automated amortised analysis of a challenging example from the literature [Atk11] .

4.1.1 Experiments

We have implemented our method in a prototype tool called Ranger. The implementation is based on the *Forester* shape analyser [HŠRV13], which represents sets of memory shapes using so-called *forest automata* (FAs). We use the *Loopus* tool [**?**] as a back-end *resource bounds analyser* for the generated integer programs. We evaluated Ranger on a set of benchmarks including programs manipulating various complex data structures or requiring amortized reasoning for inferring precise bounds.

Our experiments were performed on a machine with an Intel Core i7-2600@3.4 GHz processor and 32 GiB RAM running Debian GNU/Linux. We compared our prototype Ranger with two other tools: AProVE and Costa. For our tool, we report three times: the running time of the shape analysis of Forester (**SA**), generation of the integer program (**IG**), and bounds analysis in Loopus (**BA**). For the other tools, we report times as reported by their web interface[6]. Further, from the outputs of the tools, we extracted the reported complexity of the main program loop, and, if needed, simplified the bounds to the big $\mathcal{O}$ notation.

The results are summarized in Table 4. We use TIMEOUT(60s) if a

[6]We could not directly compare the tools on the same machine due to the tool availability issues.

time-out of 60 seconds was hit, ERROR if the tool failed to run the example[7], and UNKNOWN if the tool could not bound the main loop of the example.

In benchmarks marked with (*), APROVE returned an incorrect bound in our experiments. Further, in benchmarks marked with (**), we obtained different bounds from different runs of APROVE even though it was run in exactly the same way. In both cases, we were unable to find the reason.

The results confirm that our approach, conceived as highly parametric in the underlying shape and bounds analyses, allowed us to successfully combine an advanced shape analysis with a state-of-the-art implementation of amortized resource bounds analysis. The most encouraging result is the fully automatically computed precise linear bound for the `mergeInner` method [Atk11].

5 Conclusion and Future Directions

The main goal of this thesis was to improve the state of the art of formal analysis and verification of systems with infinite state space and with the focus on techniques based on automata. In particular, we addressed this goal in two distinct parts.

In the first part, we focused on weak monadic second-order logic of one successor (WS1S): a highly expressive, yet decidable, theory that was successfully applied in several formal analyses and verification methods. First, we limited ourselves to formulae in the prenex normal form and proposed an antichain-based decision procedure. The procedure checks the validity of a formula by constructing a corresponding finite automaton for the matrix (i.e. the quantifier free sub-formula) of the given formula followed by processing the prefix of quantifiers by recursively computing the fixpoint of final resp. nonfinal states. Finally, the method concludes that the formula is valid if the intersection of initial and final states is non-empty. We further optimized this procedure by a generalization of the antichain-based universality checking which allows us to considerably reduce the explored state space. We demonstrated the efficiency of this method on a series of both artificial formulae and formulae describing invariants of programs manipulating with singly-linked lists beating the state-of-the-art methods.

We further generalized the procedure to arbitrary formulae. We proposed a systematic way to express the formulae as so-called language terms and check validity of the formulae using an on-the-fly algorithm. We optimized the basic algorithm by two main techniques: a antichain-based pruning of the state space and a lazy evaluation of the sub-terms. We evaluated the procedure on a series of formulae used for verification of programs manipulating singly-linked lists or

[7]However, we verified that all our examples are syntactically correct.

arrays. Our second procedure outperformed both the state-of-the-art methods as well as our initial approach by several orders of magnitude.

In the second part of the thesis, we focused on resource bounds analysis of heap-manipulating programs. We proposed a novel parametric class of shape norms that express the distance between two distinct points through selector paths (such as the norm $x\langle \text{next}^* \rangle \texttt{NULL}$ expressing the lengths of paths through the `next` selector from the variable x to null pointer). Based on this class of norms, we designed a method that transforms an input heap-manipulating program into a corresponding integer representation. We then analyse the resulting integer program using state-of-the-art resource bounds analysers for integer programs. In order to construct the integer representation efficiently, we propose to (1) derive the norms directly from the program, (2) use a calculus that infers changes of norms according to the results of the shape analysis, and (3) prune the set of tracked norms based on several heuristics. We evaluated our approach on a series of programs either manipulating non-trivial data structures or requiring amortized reasoning for inferring precise resource bounds. Our procedure managed to outperform the state-of-the-art methods both in terms of the speed and the precision of the bounds.

All of our contributions were implemented as tools. The first antichain based method was implemented as a prototype tool called dWiNA [FHLV14] and the second lazy method was implemented in the Gaston tool [FHJ+16]. We implemented the novel resource bounds analyser Ranger [FHR+18b] on top of the Forester and Loopus tools.

5.1 Further Directions

In the introduction, we discussed that the current state of the art of performance analysis of complex data structures is still less developed than, e.g. analysis of integer programs. While our contributions have hopefully pushed the usability border of both performance analysis and WS1S logic, we still think that there is a lot of potential directions we could follow or enhance.

We mentioned that WS1S has many applications, e.g, as an underlying theory for specification of invariants of linear data structures. The next natural step is to extend our procedures to WS2S — weak monadic second-order logic of two successors — which would enable us to model properties of more complex data structures, such as trees. However, one will have to cope with a more complex type of automata, in particular, tree automata. For tree automata, however, even some basic operations, such as subsumption testing or the simulation relation, are more complex and more expensive. Moreover, while our procedures performed well on many formulae, e.g. describing properties of arrays [ZHW+14] or singly-linked lists [MPQ11], they still failed on

many other due to a state space explosion — an inherent property of WS1S. We believe that we could achieve further state space reduction by adapting, e.g. simulation-based techniques [Cé17] both on the generated automata as well as by weakening the term-subsumption relation. Another possible reduction could be achieved by integration of our approach with SAT and SMT techniques or by adapting some of the techniques proposed by the authors of the MONA tool [KMS02].

In the second part, we proposed a resource bounds analysis of the heap-manipulating programs based on a new class of norms, which model numerical measures such as lengths of lists by selector paths between distinct points. Our class of norms is currently limited to simple selector paths only, and so we would like to extend its calculus to a richer variety of selector paths. In particular, we would like to support concatenation (to support sequential traversals in the control flow) and iteration (to support nested cycles in the control flow) of selector paths. Moreover, we believe we could infer resource bounds for a wider class of programs if we combined our path-based norms with size-based norms as defined, e.g. in APROVE [FG17]. At last, we would like to extend our approach to analysis of open programs. We believe that adapting bi-abduction techniques [LQC15] for resource bounds analysis could allow us to scale better and to be applied in the practice.

In this thesis, we mainly researched possibilities of static analysis for performance analysis of programs. Another direction that we are currently exploring is dynamic analysis of programs. In particular, we propose to model the performance of programs based on real data captured from actual program runs. One can then model the performance of a program using, e.g. regression, non-parametric, or multivariate analyses [Dev11] . Based on these models, we believe we can automatically detect performance changes between two distinct versions of programs, i.e. detect from pairs of models for current and some baseline version of a program that the performance considerably degraded. Moreover, we wish to explore possibilities of using, e.g. fuzz-testing for triggering performance changes in program runs. Finally, we would like to develop an optimized collection of resource data from real runs by limiting the analysis only to subset of program units (functions, etc.) that impacts the program performance the most. Of course, a question is how to find this subset.

5.2 Publications Related to this Thesis

We developed two decision procedures for WS1S logic. The first one based on antichains was initially published in TACAS'15 [FHLV15]; its extended version with additional proofs and more thorough examples was published in the Acta Informatica journal [FHLV19]. Our follow-up work, which generalised the

original decision procedure to arbitrary formulae based on lazy techniques and on-the-fly exploration of the state space was published in TACAS'17 [FHJ+17].

In the field of performance analysis of heap-manipulating programs, we built on efficient approaches of the Loopus and Forester tools and proposed a parametric framework for amortized resources bounds analysis published in VMCAI'18 [FHR+18a].

In summary, we developed three tools. The implementations of antichain based and lazy decision procedures for WS1S logic called dWiNA [FHLV14] and Gaston [FHJ+16] respectively. Further, we extended the Forester tool into a Ranger tool [FHR+18b] which translates input heap-manipulating programs into corresponding integer programs. At last, our ongoing work is currently developed as the Perun tool [FGL+18].

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*. 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*. Springer, 2010.

[Atk11] Robert Atkey. Amortised resource analysis with separation logic. *Logical Methods in Computer Science*, 2011.

[Büc59] Julius Richard Büchi. Weak second-order arithmetic and finite automata. Technical report, The University of Michigan, 1959.

[Cé17] G. Cécé. Foundation for a series of efficient simulation algorithms. In *In Proc. of LICS'17*. IEEE, June 2017.

[Dev11] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole, 2011.

[DR10] Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In *Proc. of TACAS'10*. Springer, 2010.

[DV] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *In Proc. of POPL'14*. ACM.

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

[FG17] Florian Frohn and Jürgen Giesl. Complexity analysis for Java with AProVE. In *Proc of IFM'17*. Springer, 2017.

[FGL+18] Tomáš Fiedor, Martina Grzybowská, Matuš Liščinský, Jiří Pavela, Radim Podola, and Šimon Stupinský. `Perun`, 2018. Available from `https://github.com/tfiedor/perun`.

[FHJ+16] Tomáš Fiedor, Lukás Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. GASTON, 2016. Available from `http://www.fit.vutbr.cz/research/groups/verifit/tools/gaston/`.

[FHJ+17] Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. Lazy automata techniques for WS1S. In *Proc. of TACAS'17*. Springer Verlag, 2017.

[FHLV14] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. `dWiNA`, 2014. Available from `http://www.fit.vutbr.cz/research/groups/verifit/tools/dWiNA/`.

[FHLV15] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Nested antichains for WS1S. In *Proc. of TACAS'15*. Springer, 2015.

[FHLV19] Tomáš Fiedor, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Nested antichains for WS1S. *Acta Informatica*, 2019.

[FHR+18a] Tomáš Fiedor, Lukáš Holík, Adam Rogalewicz, Moritz Sinn, Tomáš Vojnar, and Florian Zuleger. From shapes to amortized complexity. In *Proc. of VMCAI'18*. Springer, 2018.

[FHR+18b] Tomáš Fiedor, Lukáš Holík, Adam Rogalewicz, Moritz Sinn, Tomáš Vojnar, and Florian Zuleger. RANGER, 2018. Available from `http://www.fit.vutbr.cz/research/groups/verifit/tools/ranger/`.

[GK10] Tobias Ganzow and Lukasz Kaiser. New algorithm for weak monadic second-order logic on inductive structures. In *Proc. of CSL'10*. Springer, 2010.

[HŠRV13] Lukáš Holík, Jiří Šimáček, Adam Rogalewicz, and Tomáš Vojnar. Fully Automated Shape Analysis Based on Forest Automata. In *Proc. of CAV'13*, LNCS. Springer, 2013.

[JSS+12] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proc. of PLDI'12*, 2012.

[KMS02] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 2002.

[LQC15] Ton Chanh Le, Shengchao Qin, and Wei Ngan Chin. Termination and non-termination specification inference. In *Proc. of PLDI'15*. ACM, 2015.

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

[Mey72] Albert R. Meyer. Weak monadic second order theory of successor is not elementary-recursive. In *Proc. of Logic Colloquium*. Springer, 1972.

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

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

[SZV17] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 2017.

[Tra15] Dmitriy Traytel. A coalgebraic decision procedure for WS1S. In *CSL'15*. Schloss Dagstuhl, 2015.

[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*. Springer, 2006.

[ZHW+14] Min Zhou, Fei He, Bow-Yaw Wang, Ming Gu, and Jiaguang Sun. Array theory of bounded elements and its applications. *Journal of Automated Reasoning*, 2014.

Curriculum Vitae

Personal Data

Name:	Tomáš Fiedor
Nationality:	Czech Republic
Date of birth:	May 1, 1990
E-mail:	`ifiedortom@fit.vutbr.cz`
Homepage:	`http://www.fit.vutbr.cz/~ifiedortom/`

Education

since 2014	Brno University of Technology, Faculty of Information Technology, studying in Ph.D. study programme Computer Science and Engineering
2012 – 2014	Brno University of Technology, Faculty of Information Technology, master's degree in Information Technology, master's thesis *A Decision Procedure for the WSkS Logic*
2009 – 2012	Brno University of Technology, Faculty of Information Technology, bachelor's degree in Information Technology, bachelor thesis *Překladač jazyka C v prostředí Python*
2001 – 2009	Gymnázium Orlová, eight year grammar school

Accomplishments

6/2014	Dean's prize for master's thesis
5/2014	EEICT student competition, Brno, paper *A Decision Procedure for the WSkS Logic*, third place in category *Intelligent Systems*, master section
6/2012	Dean's prize for bachelor's thesis
5/2012	EEICT student competition, Brno, paper *C Compiler in Python*, first place in category *Computer Systems*, bachelor section

Research Activities

since 2014 Member of VeriFIT, a research group at Faculty of Information Technology Brno University of Technology that focuses on formal verification. A particular focus on 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 various approaches to performance analysis.

Language skills

Czech, English.

Abstrakt

Tato práce se věnuje vylepšení současného stavu formalní analýzy a verifikace založené na automatech a zaměřené na systémy s nekonečnými stavovými prostory.

V první části se práce zabývá dvěma rozhodovacími procedurami pro logiku WS1S, které jsou založené na korespondenci mezi formulemi logiky WS1S a konečnými automaty. První metoda je založena na tzv. antiřetězcích, ale, je limitována pouze na formule v prenexním normálním tvaru. Následně je tento přístup zobecněn na libovolné formule, jsou zavedeny tzv. jazykové termy a na jejich základě je navržena nová procedura, která pracuje za běhu a zpracovává tyto termy "líným" způsobem. Abychom získali efektivní rozhodovací proceduru, je dále navržena sada optimalizací (přičemž některé nejsou limitovány pouze pro naše přístupy). Obě metody jsou srovnány s ostatními nástroji implementujícími různé známé rozhodovací procedury. Získané výsledky jsou povzbuzující a ukazují, že použitelnost logiky WS1S je možno rozšířit na širší třídu formulí.

V druhé části se práce zabývá analýzou mezí zdrojů programů manipulujících s haldou. Je zde navržena nová třída tzv. tvarových norem založených na délkách cest mezi význačnými místy na haldě, které jsou automaticky odvozovány z analyzovaného programu. Na základě této třídy norem je dále navržen kalkul, který je schopen přesně odvodit změny odvozených normů a použít je k vygenerování odpovídající celočíselné reprezentace vstupního programu, která je následně využita pro následovanou dedikovanou analýzou mezí zdrojů. Tato metoda byla implementována nad analýzou tvaru založenou na tzv. lesních automatech, implementovanou v nástroji Forester, a dále byl použit dobře zavedený analyzátor mezí zdrojů, implementovaný v nástroji Loopus. V experimentální evaluaci bylo ukázáno, že je opravdu takto získán silný analyzátor, který je schopen odvodit meze programů, které ještě nikdy plně automatizovaně odvozené nebyly.