VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

A DECISION PROCEDURE FOR THE WSKS LOGIC

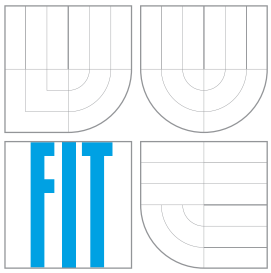
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ FIEDOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ROZHODOVACÍ PROCEDURA PRO LOGIKU WSKS

A DECISION PROCEDURE FOR THE WSKS LOGIC

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ FIEDOR

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ONDŘEJ LENGÁL

BRNO 2014

Abstrakt

Různé typy logik se často používají jako prostředky pro formální specifikaci systémů. Slabá monadická logika druhého řádu s k následníky (WSkS) je jednou z nich a byť má poměrně velkou vyjadřovací sílu, stále je rozhodnutelná. Ačkoliv složitost testování splnitelnosti WSkS formule není ani ve třídě ELEMENTARY, tak existují přístupy založené na deterministických automatech, implementované např. v nástroji MONA, které efektně řeší omezenou třídu praktických příkladů, nicméně nefungují pro jiné. Tato práce rozšiřuje třídu prakticky řešitelných příkladů, a to tak, že využívá nedávno vyvinutých technik pro efektní manipulaci s nedeterministickými automaty (jako je například testování universality jazyka pomocí přístupu založeného na antichainech) a navrhuje novou rozhodovací proceduru pro WSkS využívající právě nedeterministické automaty. Procedura je implementována a ve srovnání s nástrojem MONA dosahuje v některých případech řádově lepších výsledků.

Abstract

Various types of logics are often used as a means for formal specification of systems. The weak monadic second-order logic of k successors (WSkS) is one of these logics with quite high expressivity, yet still decidable. Although the complexity of checking satisfiability of a WSkS formula is not even in the ELEMENTARY class, there are approaches to this problem based on deterministic tree automata that perform well in practice, like the MONA tool that efficiently solves the class of practical formulae, but fails for some others. This work extends the class of practically solvable formulae with the use of recently developed techniques for efficient manipulation of non-deterministic automata (such as the antichains algorithm for testing universality) and designs a new decision procedure using non-deterministic automata. The procedure is implemented and is compared with the MONA tool and for some cases yield better results than MONA.

Klíčová slova

formální verifikace, stromové automaty, WSkS, rozhodovací procedury

Keywords

formal verification, tree automata, WSkS, decision procedures

Citace

Tomáš Fiedor: A Decision Procedure for the WSkS Logic, diplomová práce, Brno, FIT VUT v Brně, 2014

A Decision Procedure for the WSkS Logic

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana inženýra Ondřeje Lengála. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Fiedor
23. května 2014

Poděkování

Děkuji vedoucímu práce, Ing. Ondřeji Lengálovi, za odborné vedení, ochotu konzultovat, motivaci a především za trpělivost. Dále bych chtěl poděkovat Prof. Ing. Tomáši Vojnarovi, Ph. D. a Mgr. Lukáši Holíkovi, Ph. D. za poskytnuté konzultace a sezení. Rovněž bych chtěl poděkovat bratrovi Ing. Janu Fiedorovi za veškeré diskuze a rodině za podporu ve studiu.

Contents

Chapter 1

Introduction

The use of logics has always had an important place in various sciences, especially in computer science and, in particular, formal verification. The expressiveness of logics allows us to specify verified systems in a very natural and intuitive way without the need of deep knowledge of the used verification procedures. The logic used range from plain propositional logic through all kinds of first-order logics like Presburger arithmetic [21], separation logic, and all the way to the more complex monadic second-order logics, like WSkS [5]. However, with great expressiveness comes great complexity of decision procedures of these logics, ranging from NP-complete for propositional logic, through PSPACE-complete for quantified Boolean formulae, with some stronger logics not being decidable at all.

WSkS stands for *weak monadic second-order logic of k successors*. Roughly, this means that it allows to quantify over finite set variables where every element from the universe of discourse has k successors. This properties open ways for expressing various k-ary tree structures, e.g. binary trees or heaps, and linear structures, e.g. linked lists, as well. Decision procedures for WSkS are usually based on the correspondence between WSkS formulae and languages of finite automata (be it word or tree automata). The atomic subformulae of an examined WSkS formula ϕ are translated to finite (word/tree) automata, which are further connected or manipulated using automata manipulation techniques according to the structure of ϕ. The resulting automaton then represents the language encoding all models of ϕ. The problem of checking satisfiability (unsatisfiability/validity/invalidity) of ϕ is in the NONELEMENTARY complexity class [20]. The main source of this complexity is the occurence of quantifier alternations in ϕ. In the automata construction each quantifier alternation yields projection and complementation of the automaton, for which there is currently no known algorithm other than exponential.

There has been several attempts to implement a decision procedure for WSkS, like [10] for $k = 1$. Currently the best one is the tool MONA [11], an implementation of an automata-based decision procedure which is quite fast and uses deterministic finite word and binary bottom-up tree automata for deciding WS1S and WS2S formulae respectively. The authors of MONA have developed a number of heuristics, such as the use of binary decision diagrams for the representation of transition functions of automata or cache-friendly implementation of hash tables that made MONA perform well on many practical examples in spite of the terrifying worst-case complexity. However, there still remains a large class of practical examples for which MONA fails.

Recently, there has been a major advance in the algorithms manipulating non-deterministic automata (like [18]). Even problems with high worst-case complexity like testing universality or language inclusion of the automaton can now be solved efficiently in many practical cases using algorithms that heuristically prune the search space, such as algorithms based on antichains or on the simulation relation among states of automata.

This text describes the design and implementation of a new decision procedure for WSkS that uses non-deterministic tree automata while exploiting the techniques for their efficient manipulation to achieve better performance.

Tha main idea of the decision procedure is to transform a WSkS formula ϕ into equivalent formula in the form of $\phi' = \neg\exists\mathcal{X}_n\neg\ldots\neg\exists\mathcal{X}_1 : \psi$, where ψ (called the *matrix*) is a quantifier free formula constructed over atomic formulae and their negations, using only propositional connectives $\wedge$ and $\vee$. The procedure then constructs the automaton $\mathcal{A}_\psi$ corresponding to ψ and checks satisfiability of ϕ' while working only with $\mathcal{A}_\psi$ without explicitly constructing the automaton for ψ'.

The rest of this thesis is organized as follows. In Chapter 2 all necessary preliminaries are defined. A short introduction to the theory of formal languages will describe finite word and tree automata, as well as some properties of their languages. Another structure that will be defined there will be Binary Decision Diagrams, or BDDs for short, and their extensions. Chapter 3 defines the syntax and semantics of the WSkS logic and its restricted forms. A brief description of the correspondence between tree automata and WSkS formulae also appears there. Chapter 4 describes the approach of MONA and all the known tweaks and secrets that were used during its development. In Chapter 5 our approach to a decision procedure for the WSkS logic using non-deterministic automata and its antichain-based principle will be outlined. A short introduction to antichain-based procedures and universality testing of finite word and tree automata is described there as well. Chapter 6 describes the implementation of the designed algorithm as well as some used optimizations. The implementation is then evaluated in Chapter 7 and compared with MONA in various aspects like the speed time or the size of the generated state space. Chapter 8 summarizes this thesis.

Chapter 2

Preliminaries

2.1 Formal Languages

We define an *alphabet* as a finite non-empty set of elements called *symbols*. A *word* over the alphabet Σ is a finite sequence $a_1 a_2 \dots a_n$, such that $a_i \in \Sigma$, for all $1 \leq i \leq n$. The *empty sequence* of symbols, i.e. a sequence which does not contain any symbols, is denoted as ϵ.

Let $x = a_1 a_2 \dots a_n$ and $y = b_1 b_2 \dots b_m$ be words over the alphabet Σ, for some $n, m \in \mathbb{N}$. The *concatenation* of words x and y is defined as the word $xy = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$. Note that $\epsilon x = x\epsilon = x$.

Let Σ be an alphabet. We denote the set of all words over Σ as Σ^*. The set of all words except for the empty word is denoted as $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$. We call $L \subseteq \Sigma^*$ a *language* over Σ.

2.2 Finite Automata

A *non-deterministic finite* (word) *automaton* (further abbreviated as FA) is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, where

- Q is a finite set of *states*,
- Σ is the input alphabet,
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. We use $p \xrightarrow{a} q$, for $p, q \in Q$ and $a \in \Sigma$ to denote that $(p, a, q) \in \delta$,
- $I \subseteq Q$ is the set of *initial states*,
- $F \subseteq Q$ is the set of *final states*.

Further we call a set of states in $\mathcal{A}$, i.e. a subset of Q, a *macro-state* and we define the post-image of a state p as $Post(p) = \{p' \in Q \mid \exists a \in \Sigma : (p, a, p') \in \delta\}$.

Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a FA. A *run* of $\mathcal{A}$ over the word $w = a_1 a_2 \dots a_n \in \Sigma^*$ from the state $p \in Q$ to the state $r \in Q$ is a sequence of states $q_0 q_1 \dots q_n$, such that $q_0 = p, q_n = r$ and for all $1 \leq i \leq n$ there is a transition $q_{i-1} \xrightarrow{a_i} q_i$ in δ. We write $p \overset{w}{\Longrightarrow} r$ to denote that there exists a run from the state p to the state r over the word w.

The *language* accepted by a state $q \in Q$ is defined as $L_{\mathcal{A}}(q) = \{w \mid q \overset{w}{\Longrightarrow} q_f, q_f \in F\}$. If it is clear which FA $\mathcal{A}$ we are referring to, we can simplify this to $L(q)$. The language accepted by a set of states $S \subseteq Q$ is further defined as $L_{\mathcal{A}}(S) = \bigcup_{q \in S} L_{\mathcal{A}}(q)$ and the language accepted by the automaton $\mathcal{A}$ is defined as $L(\mathcal{A}) = L_{\mathcal{A}}(I)$.

A *deterministic finite automaton* (DFA) is a FA $\mathcal{A}$ where $|I| = 1$ and $\forall q \in Q, \forall a \in \Sigma : |\{r \in Q \mid q \xrightarrow{a} r\}| \leq 1$, i.e. δ is a partial function $\delta : Q \times \Sigma \longrightarrow Q$. If δ is total, i.e. $\forall q \in Q, \forall a \in \Sigma : |\delta(q, a)| = 1$, we call $\mathcal{A}$ a *complete deterministic finite automaton*. It can be shown that for every deterministic FA there exists a language equivalent complete deterministic FA, by adding a new non-final sink state.

Lemma 2.1. *For every non-deterministic finite automaton $\mathcal{A}$, there exists a deterministic finite automaton $\mathcal{A}'$ such that $L(\mathcal{A}) = L(\mathcal{A}')$.*

Proof. Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a FA. We can construct the DFA $\mathcal{A}' = (Q', \Sigma, \delta', I', F')$, such that $L(\mathcal{A}) = L(\mathcal{A}')$, by the following method:

$$\begin{aligned} Q' &= 2^Q, \\ \delta' &= \{S \xrightarrow{a} R \mid S \in 2^Q, R = \{r \mid \exists s \in S : s \xrightarrow{a} r\}\}, \\ I' &= \{I\}, \\ F' &= \{S \in 2^Q \mid S \cap F \neq \emptyset\}. \end{aligned}$$

Note that $\mathcal{A}'$ is complete. It can be proved that $L(\mathcal{A}) = L(\mathcal{A}')$ by showing that $L(\mathcal{A}) \subseteq L(\mathcal{A}')$ and simultaneously $L(\mathcal{A}') \subseteq L(\mathcal{A})$ [19]. □

Definition 2.1. *We define the class of regular languages $\mathcal{L}_R$ as the class of languages $L \in \Sigma^*$ such that there exists a finite automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*

Example 2.1. *Consider the encoding of subsets of $\{1, \ldots, n\}$, for some $n \in \mathbb{N}$, as binary strings $X = a_1 \ldots a_n$, where a_i is 1 if $i \in X$ and 0 if $i \notin X$. We can construct the automaton $\mathcal{A}$ that accepts the encoding of the set difference X of two sets Y and Z, i.e. $X = Y \setminus Z$, as depicted in Figure 2.1.*

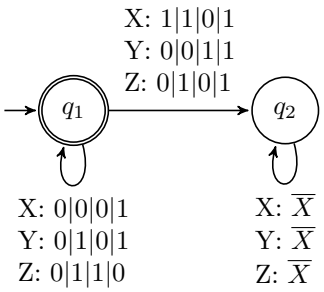


Figure 2.1: Automaton $\mathcal{A}$ accepting the encoding of the set difference of a pair of sets. Note that we use the symbol $\overline{X}$ as don't-care symbol, i.e. it can stand for any symbol from Σ.

2.2.1 Closure Properties of Regular Languages

Theorem 2.1. *The class of regular languages is closed under union.*

Proof. Let $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$ be a pair of finite automata such that $Q_\mathcal{A} \cap Q_\mathcal{B} = \emptyset$. We construct an automaton $\mathcal{A} \cup \mathcal{B}$ accepting the *union* of languages $L(\mathcal{A})$ and $L(\mathcal{B})$:

$$\mathcal{A} \cup \mathcal{B} = (Q_\mathcal{A} \cup Q_\mathcal{B}, \Sigma, \delta_\mathcal{A} \cup \delta_\mathcal{B}, I_\mathcal{A} \cup I_\mathcal{B}, F_\mathcal{A} \cup F_\mathcal{B}). \tag{2.1}$$

The proof that $L(\mathcal{A} \cup \mathcal{B}) = L(\mathcal{A}) \cup L(\mathcal{B})$ can be found for example in [19]. □

Theorem 2.2. *The class of regular languages is closed under intersection.*

Proof. Let $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$ be a pair of finite automata. We construct an automaton $\mathcal{A} \cap \mathcal{B}$ accepting the *intersection* of languages $L(\mathcal{A})$ and $L(\mathcal{B})$:

$$\mathcal{A} \cap \mathcal{B} = (Q_\mathcal{A} \times Q_\mathcal{B}, \Sigma, \delta, I_\mathcal{A} \times I_\mathcal{B}, F_\mathcal{A} \times F_\mathcal{B}) \tag{2.2}$$

where

$$\delta = \{(p_1, p_2) \xrightarrow{a} (q_1, q_2) \mid p_1 \xrightarrow{a} q_1 \in \delta_\mathcal{A} \wedge p_2 \xrightarrow{a} q_2 \in \delta_\mathcal{B}\}. \tag{2.3}$$

The proof that $L(\mathcal{A} \cap \mathcal{B}) = L(\mathcal{A}) \cap L(\mathcal{B})$ can be found for example in [19]. □

Theorem 2.3. *The class of regular languages is closed under language complementation.*

Proof. Let $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ be a complete deterministic FA. We construct an automaton accepting the *complement* of $L(\mathcal{A})$:

$$\overline{\mathcal{A}} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, Q_\mathcal{A} \setminus F_\mathcal{A}). \tag{2.4}$$

The proof that $L(\overline{\mathcal{A}}) = \Sigma^* \setminus L(\mathcal{A})$ can be found for example in [19]. □

2.3 Tree Automata

A *ranked alphabet* Σ is a finite set of symbols together with a ranking function $\# : \Sigma \to \mathbb{N}$, we call $\#a$ the *rank* of a. For any $n \geq 0$, we denote by Σ_n the set of all symbols of rank n from Σ. We denote by $\epsilon \in \mathbb{N}^*$ the *empty sequence*.

Then a *tree* t over a ranked alphabet Σ is defined as a partial mapping $t : \mathbb{N}^* \to \Sigma$, that satisfies the following conditions:

1. $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$,
2. for every $v \in dom(t)$, called a *node* of t, the following holds: $(\#t(v) = n \geq 0) \Longrightarrow \{i \mid vi \in dom(t)\} = \{1, \ldots, n\}$.

For a node v, the i-th *child* of v is the node vi, and the i-th *subtree* of v is the tree t' such that for all $v' \in \mathbb{N}^*$, $t'(v') = t(viv')$. A node v which does not have any children is called a *leaf* of the tree t. The set of all trees over the alphabet Σ is denoted as T_Σ.

A (finite, non-deterministic) *tree automaton* (further abbreviated as TA) is a quadruple $\mathcal{A} = (Q, \Sigma, \delta, F)$, where:

- Q is a finite set of *states*,
- Σ is a *ranked alphabet*,
- δ is the *set of transitions*,
- $F \subseteq Q$ is the set of *final states*.

Each transition is defined as a triple $((q_1, \ldots, q_n), a, q)$, where $q_1, \ldots, q_n, q \in Q, a \in \Sigma$ and $\#a = n$. We use equivalently $(q_1, \ldots, q_n) \xrightarrow{a} q$ and $q \xrightarrow{a} (q_1, \ldots, q_n)$ to denote that $((q_1, \ldots, q_n), a, q) \in \delta$, for *bottom-up* and *top-down* representation respectively. In the special case where $n = 0$, we speak about the so-called *leaf rules* that can be abbreviated as $\xrightarrow{a} q$ or $q \xrightarrow{a}$.

Let $\mathcal{A} = (Q, \Sigma, \delta, F)$ be a TA. We define a *run* of $\mathcal{A}$ over a tree $t \in T_\Sigma$ as a mapping $\varphi : dom(t) \to Q$ such that for each node $v \in dom(t)$ of rank $\#t(v) = n$, where $\varphi(v) = q$, if $\varphi(vi) = q_i$ for all $1 \leq i \leq n$, then $(q_1, \ldots, q_n) \xrightarrow{t(v)} q$. We write $t \xRightarrow{\varphi} q$ to denote that φ is a run of $\mathcal{A}$ over t such that $\varphi(\epsilon) = q$. We write $t \Longrightarrow q$ to denote that there exists a run φ for which $t \xRightarrow{\varphi} q$.

The *language* accepted by a state q is defined as $L_\mathcal{A}(q) = \{t \mid t \Longrightarrow q\}$. For a set of states $S \subseteq Q$ we define the language accepted by this set as $L_\mathcal{A}(S) = \bigcup_{q \in S} L_\mathcal{A}(q)$. Similarly to FA, if it is clear which TA $\mathcal{A}$ we are referring to, we only write $L(q)$ or $L(S)$. Then language of $\mathcal{A}$ is defined as $L(\mathcal{A}) = L_\mathcal{A}(F)$.

Definition 2.2. *A* deterministic finite tree automaton *(abbreviated as DTA) is a TA such that there are no two rules with the same left-hand, or right-hand, side in δ for bottom-up DTA or top-down DTA respectively.*

Note that the expressive power of bottom-up and top-down NTA is the same. However, top-down DTA are strictly less powerful than top-down NTA. See [6] for more details.

Definition 2.3. *We define the class of regular tree languages $\mathcal{L}$ as the class of languages $L \subseteq T_\Sigma$ such that there exists a finite tree automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$.*

2.3.1 Closure Properties of Regular Tree Languages

Theorem 2.4. *The class of regular tree languages is closed under intersection.*

Proof. Let $\mathcal{A} = (Q_\mathcal{A}, \Sigma, F_\mathcal{A}, \delta_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, F_\mathcal{B}, \delta_\mathcal{B})$ be two tree automata. We construct a tree automaton $\mathcal{A} \cap \mathcal{B}$ accepting the *intersection* of languages $L(\mathcal{A})$ and $L(\mathcal{B})$:

$$\mathcal{A} \cap \mathcal{B} = (Q_\mathcal{A} \times Q_\mathcal{B}, \Sigma, F_\mathcal{A} \times F_\mathcal{B}, \delta) \tag{2.5}$$

where

$$\begin{aligned} \delta = \{&((q_1^1, q_1^2), \ldots, (q_n^1, q_n^2)) \xrightarrow{f} (q^1, q^2) \\ &\mid (q_1^1, \ldots, q_n^1) \xrightarrow{f} q^1 \in \delta_\mathcal{A} \wedge (q_1^2, \ldots, q_n^2) \xrightarrow{f} q^2 \in \delta_\mathcal{B})\}. \end{aligned} \tag{2.6}$$

□

Note that this construction preserves determinism, i.e. if the two given automata are deterministic, then so is the product automaton.

Theorem 2.5. *The class of regular tree languages is closed under union.*

Proof. Let $\mathcal{A} = (Q_\mathcal{A}, \Sigma, F_\mathcal{A}, \delta_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, F_\mathcal{B}, \delta_\mathcal{B})$ be two tree automata. We construct a tree automaton $\mathcal{A} \cup \mathcal{B}$ accepting the *union* of languages $L(\mathcal{A})$ and $L(\mathcal{B})$:

$$\mathcal{A} \cup \mathcal{B} = (Q_\mathcal{A} \cup Q_\mathcal{B}, \Sigma, F_\mathcal{A} \cup F_\mathcal{B}, \delta_\mathcal{A} \cup \delta_\mathcal{B}). \tag{2.7}$$

□

Theorem 2.6. *The class of regular tree languages is closed under language complementation.*

Proof. Let $\mathcal{A} = (Q, \Sigma, F, \delta)$ be a complete bottom-up deterministic TA. We construct a tree automaton accepting the *complement* of the language $L(\mathcal{A})$:

$$\overline{\mathcal{A}} = (Q, \Sigma, Q \setminus F, \delta). \tag{2.8}$$

□

Note that for non-deterministic tree automata there is currently known no complementation procedure better than first bottom-up determinizing the automaton and then complementing it using the construction above.

2.3.2 Relations on Trees

Given a ranked alphabet Σ and $n \geq 0$, let $(T_\Sigma)^n$ be the *Cartesian product* $T_\Sigma \times (T_\Sigma)^{n-1}$ with the ground case $(T_\Sigma)^0 = \{\top\}$, where $\{\top\}$ is a neutral element w.r.t. Cartesian product. A subset of $(T_\Sigma)^n$ is an n-ary relation on T_Σ. Further, let $\Sigma_\perp^n$ be the *compound alphabet* $\Sigma_\perp^n = (\Sigma \cup \{\perp\})^n$ where $\perp$ is a new symbol such that $\perp \notin \Sigma$ and $\#\perp = 0$. We write the symbol $(f_1, \ldots, f_n)$ of $\Sigma_\perp^n$ as $f_1 \ldots f_n$. Arities of symbols in $\Sigma_\perp^n$ are defined as $\#(f_1 \ldots f_n) = \max(\#f_1, \ldots, \#f_n)$.

Let $[\cdot]$ be a function that maps n-tuples of trees over T_Σ to trees over $T_{\Sigma_\perp^n}$:

$$[\cdot] : \begin{cases} (T_\Sigma)^n \to T_{\Sigma_\perp^n} \\ f_1(t_1^1, \ldots, t_1^{\#f_1}), \ldots, f_n(t_n^1, \ldots, t_n^{\#f_n}) \mapsto \\ \quad f_1 \ldots f_n([t_1^1, \ldots, t_n^1], \ldots, [t_1^m, \ldots, t_n^m]) \end{cases} \tag{2.9}$$

where m is the maximal arity of $f_1, \ldots, f_n \in \Sigma$ and t_i^j is, by convention, $\perp$ when $j > \#f_i$.

Definition 2.4. *Rec, for recognizable tree relations, is the class of relations $R \subseteq (T_\Sigma)^n$ such that the language*

$$\{[t_1, \ldots, t_n] \mid (t_1, \ldots, t_n) \in R\} \tag{2.10}$$

is accepted by a tree automaton on the alphabet $\Sigma_\perp^n$.

Proposition 2.1. *Rec is closed under Boolean operations, i.e. intersection, union and complementation.*

Proof. This is due to closure properties of tree automata (see Section 2.3.1). □

Definition 2.5. *If $R \subseteq (T_\Sigma)^n$ where $n \geq 1$ and $1 \leq i \leq n$, then the i-th* projection *of R is the relation $R_i \subseteq (T_\Sigma)^{n-1}$ defined as follows:*

$$R_i(t_1, \ldots, t_{n-1}) \Leftrightarrow \exists t \in T_\Sigma.\ R(t_1, \ldots, t_{i-1}, t, t_i, \ldots, t_{n-1}). \tag{2.11}$$

Lemma 2.2. *Rec is closed under projection.*

Proof. Let us assume that $R \in Rec$. The i-th projection R_i of R is simply its image by the following tree homomorphism:

$$h_i(f_1 \ldots f_n(t_1, \ldots, t_k)) \stackrel{def}{=} f_1 \ldots f_{i-1} f_{i+1} \ldots f_n(h_i(t_1), \ldots, h_i(t_m)) \tag{2.12}$$

where m is the arity of $(f_1 \ldots f_{i-1} f_{i+1} \ldots f_n)$, which is smaller or equal to k. Because linear homomorphisms preserve recognizability (Theorem 1.4.3 in [6]), $R_i \in Rec$. □

Definition 2.6. *If $R \subseteq (T_\Sigma)^n$ where $n \geq 0$ and $1 \leq i \leq n+1$ then the i-th* cylindrification *of R is the relation $R_i \subseteq (T_\Sigma)^{n+1}$ defined as follows:*

$$R^i(t_1, \ldots, t_{i-1}, t, t_i, \ldots, t_n) \Leftrightarrow R(t_1, \ldots, t_{i-1}, t_i, \ldots, t_n). \tag{2.13}$$

Lemma 2.3. *Rec is closed under cylindrification.*

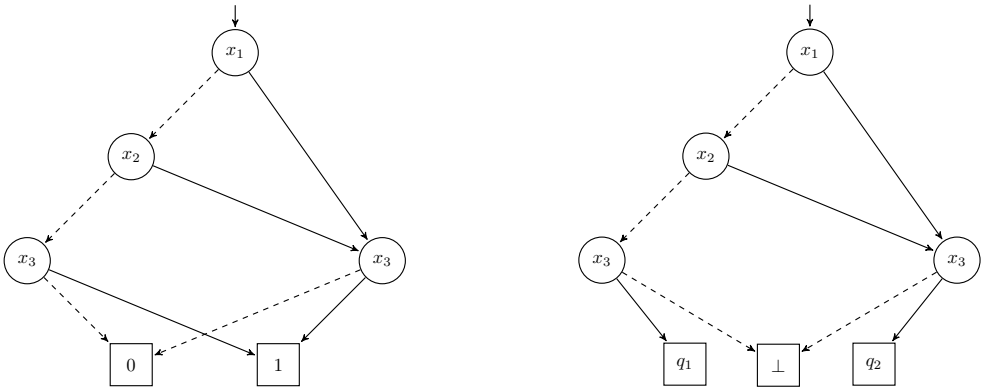
Proof. Similarly to projection, i-th cylindrification is obtained as an inverse homomorphic image, and thus is recognizable as stated by Theorem 1.4.4. in [6]. □

2.4 Binary Decision Diagrams

We define a *Boolean function* of *arity* k as a function $f : \{0,1\}^k \to \{0,1\}$. A *reduced ordered binary decision diagram* (abbreviated as ROBDD or just BDD) r over a set of n Boolean variables $X = \{x_1, \ldots, x_n\}$ is a connected directed acyclic graph with a single *source node* called *root* and at least one of two sink nodes 0 and 1. Nodes that are not sink nodes are called *internal nodes*. Assignment of Boolean variables to each of the internal nodes is done by the function *var*. We assume that X is ordered in the following way: $x_1 < x_2 < \ldots < x_n$. For every internal node v, there exists two outgoing edges labeled as *low* and *high*, such that $var(v) < var(v.low) \wedge var(v) < var(v.high)$; and $v.low \neq v.high$ as well (since otherwise it could be further reduced).

Nodes of a BDD represent n-ary Boolean functions that map each assignment to the Boolean variables in X to a corresponding Boolean value defined as follows, using $\overline{x}$ as an abbreviation for $x_1 \ldots x_n$:

$$\begin{array}{lcl} [\![0]\!] & = & \lambda\overline{x}.0 \\ [\![1]\!] & = & \lambda\overline{x}.1 \\ [\![v]\!] & = & \lambda\overline{x}.(\neg x_i \wedge [\![v.low]\!](\overline{x})) \vee (x_i \wedge [\![v.high]\!](\overline{x})) \\ & \text{where} & var(v) = x_i \end{array}$$



(a) BDD (b) MTBDD

Figure 2.2: Difference between BDD and MTBDD.

We can further extend the notion of these functions to an arbitrary nonempty codomain S, $f : \{0,1\}^k \longrightarrow S$. The notion of ROBDDs can then be further generalized to *multi-terminal binary decision diagrams* (abbreviated as MTBDDs), which is essentially the same data structure as a BDD, with the only difference being the fact that the set of sink nodes is not restricted to only two nodes but to any number of sink nodes labelled by elements of S. All standard notions for ROBDDs can be naturally extended to MTBDDs.

A *shared* MTBDD s is a MTBDD with multiple source nodes (or roots) that represent a mapping of every element of the set of roots R to a function induced by the MTBDD corresponding to the given root. We can see the difference between MTBDDs and BDDs in the Figure 2.2.

For manipulation with BDDs f, g we define the $Apply_1$ function for some unary leaf operator op_1 and the $Apply_2$ function for some binary leaf operator op_2 as follows:

$$Apply_1(f, op_1) = \lambda\overline{x}\,.\,op_1([\![f(\overline{x})]\!]), \tag{2.14}$$

$$Apply_2(f, g, op_2) = \lambda\overline{x}\,.\,op_2([\![f(\overline{x})]\!], [\![g(\overline{x})]\!]). \tag{2.15}$$

2.4.1 Using Shared MTBDDs for Encoding Transition Function of Tree Automata

Let $\mathcal{A} = (Q, \Sigma, \delta, F)$ be a tree automaton, such that $\Sigma = \{0,1\}^n$, for some n. Each position $1 \leq i \leq n$ is then assigned a Boolean variable from the set $X = \{x_1, \ldots, x_n\}$. We use $Q^{\#}$ to denote the set of all tuples of states from Q with up to the maximum arity that some symbol in Σ has.

The *bottom-up* representation of the transition function δ of the TA $\mathcal{A}$ uses a shared MTBDD δ^{bu} over Σ, where the set of roots $R = Q^{\#}$, and the domain set of values of sink nodes is 2^Q that is, the MTBDD δ^{bu} represents the function $[\![\delta^{bu}]\!]$: $Q^{\#} \to (\Sigma \to 2^Q)$ where

$$[\![\delta^{bu}]\!] = \lambda(q_1, \ldots, q_p)\ a.\{q \mid (q_1, \ldots, q_p) \xrightarrow{a} q\}. \tag{2.16}$$

The *top-down* representation of the transition function δ of the TA $\mathcal{A}$ uses a shared MTBDD δ^{td} over Σ, where the set of roots $R = Q$ and the domain of labels of sink nodes is $2^{Q^{\#}}$. The MTBDD δ^{td} then represents the function $[\![\delta^{td}]\!]$: $Q \to (\Sigma \to 2^{Q^{\#}})$ where

$$[\![\delta^{td}]\!] = \lambda q\ a.\{(q_1, \ldots, q_p) \mid q \xrightarrow{a} (q_1, \ldots, q_p)\}. \tag{2.17}$$

Example 2.2. *Consider the word automaton $\mathcal{A}$ from Example 2.1 accepting the encoding of set difference of two sets with the following transitions:*

$$
\begin{aligned}
q_1 &\xrightarrow{00\overline{X}} q_1 \\
q_1 &\xrightarrow{011} q_1 \\
q_1 &\xrightarrow{110} q_1 \\
q_1 &\xrightarrow{10\overline{X}} q_2 \\
q_1 &\xrightarrow{010} q_2 \\
q_1 &\xrightarrow{111} q_2 \\
q_2 &\xrightarrow{\overline{XXX}} q_2
\end{aligned}
$$

Note that we use the symbol $\overline{X}$ to substitute either 0 or 1 in its place. The MTBDD encoding the transition function of this automaton is depicted in Figure 2.3.

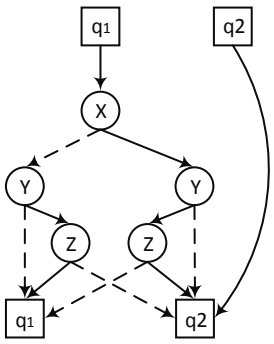


Figure 2.3: The MTBDD encoding the transition function corresponding to $\mathcal{A}$ from Example 2.1

Chapter 3

The WSkS Logic

The abbreviation WSkS stands for *weak second-order monadic logic of k successors*. This means that it is a logic that allows quantification over set variables (second-order), which can only represent *finite* sets (weak) of elements and *not functions* (monadic), over a universe of discourse where every element has k successors, and can therefore express linear (for $k = 1$) as well as tree (for $k \geq 2$) structures.

3.1 Syntax

A WSkS *term* is an empty constant ϵ, a first-order variable symbol written in lower-case letters (e.g. *x, y, z, ...*) or an unary symbol from $\{1, \ldots, k\}$ written in postfix notation. For example, $x1123$ or $\epsilon 2111$ are terms, where the latter can be shortened to 2111.

The *atomic formulae* are defined as follows:

1. For terms s and t, the equality $s = t$ is an atomic formula.
2. For terms s and t, inequalities $s \leq t$ and $s \geq t$ are atomic formulae.
3. For a term t and a second-order variable X, the membership constraint $t \in X$ is an atomic formula.

A WSkS *formula* is then built out of atomic formulae using the classical logical connectives $\wedge, \vee, \neg, \Leftarrow, \Rightarrow, \Leftrightarrow$ and quantifiers $\exists x, \forall x$ and $\exists X, \forall X$ for quantification over first-order variables and second-order variables respectively.

The syntax can be further restricted to only a subset of logical connectives and atomic formulae without harm to the expressive power of the logic. This will be further explained in Section 3.3. The set of *free variables* of a formula $\mathit{freeVars}(\psi)$ is defined as usual.

Example 3.1. *The following example WSkS formula φ denotes that the set X contains exactly one element from the universe of discourse:*

$$\varphi(X) \stackrel{def}{=} \exists p : p \in X \wedge \forall r : p \neq r \Rightarrow r \notin X \tag{3.1}$$

3.2 Semantics

We will interpret terms as strings belonging to $\{1,\ldots,k\}^*$, $=$ as the equality of strings, and $\leq$ as the prefix ordering. Second order variables will be interpreted as *finite* subsets of $\{1,\ldots,k\}^*$ with $\in$ as the membership predicate.

Let $t_1,\ldots,t_n \in \{1,\ldots,k\}^*$ and $S_1,\ldots,S_n$ be finite subsets of $\{1,\ldots,k\}^*$. Given a formula $\psi(x_1,\ldots,x_n,X_1,\ldots,X_m)$ with free variables $x_1,\ldots,x_n$, $X_n,\ldots,X_m$, the assignment $\delta = \{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n, X_1 \mapsto S_1,\ldots,X_m \mapsto S_m\}$ *satisfies* ψ written as $\delta \models \psi$ (or also $t_1,\ldots,t_n,S_1,\ldots,S_m \models \psi$) if replacing the variables with their corresponding values, the formula holds in the above model.

3.3 Restricting Syntax

We are going to restrict the WSkS syntax to use only second-order variables. This can be done by considering every first-order variable as a singleton set and transforming every formula to an equivalent one which does not contain any first-order variables. We will consider only the following atomic formulae, where X and Y are second-order variables, and build formulae over them:

- $X \subseteq Y$,
- $Sing(X)$ — holds true iff X is a singleton set,
- $X = Yi$ — holds true iff X and Y are singleton sets $\{s\}$ and $\{t\}$ respectively and $s = ti$,
- $X = \epsilon$.

We can also further simplify the syntax of WSkS formulae by restricting logical connectives used to build the formulae to only $\exists$, $\vee$, $\wedge$ and $\neg$. This syntax will be called the *restricted syntax* and its satisfaction relation will be denoted as $\models_2$.

Proposition 3.1. *There is a translation T from WSkS formulae to the restricted syntax such that*

$$s_1,\ldots,s_n,S_1,\ldots,S_m \models \psi(x_1,\ldots,x_n,X_1,\ldots,X_m) \tag{3.2}$$

if and only if

$$\{s_1\},\ldots,\{s_n\},S_1,\ldots,S_m \models_2 T(\psi)(X_{x_1},\ldots,X_{x_n},X_1,\ldots,X_m). \tag{3.3}$$

Conversely, there is a translation T' from the restricted syntax to WSkS such that

$$S_1,\ldots,S_m \models T'(\psi)(X_1,\ldots,X_m) \tag{3.4}$$

if and only if

$$S_1,\ldots,S_m \models_2 \psi(X_1,\ldots,X_m). \tag{3.5}$$

Proof. We will only present a short sketch of the proof for this proposition. For further details see [6].

We can suppose that formulae will be built only upon the atomic formulae $t \in X$ and $s = t$ and so flatten the rest of the atomic formulae. Every first-order variable y will be mapped to a second-order variable X_y as follows:

$$T(y \in X) \stackrel{def}{=} X_y \subseteq X \tag{3.6}$$
$$T(y = xi) \stackrel{def}{=} X_y = X_x i \tag{3.7}$$
$$T(x = \varepsilon) \stackrel{def}{=} X_x = \varepsilon \tag{3.8}$$
$$T(x = y) \stackrel{def}{=} X_x = X_y \tag{3.9}$$
$$T(\psi \vee \phi) \stackrel{def}{=} T(\psi) \vee T(\phi) \tag{3.10}$$
$$T(\psi \wedge \phi) \stackrel{def}{=} T(\psi) \wedge T(\phi) \tag{3.11}$$
$$T(\neg\phi) \stackrel{def}{=} \neg T(\phi) \tag{3.12}$$
$$T(\exists X.\phi) \stackrel{def}{=} \exists X.T(\phi) \tag{3.13}$$
$$T(\exists y.\phi) \stackrel{def}{=} \exists X_y : \ Sing(X_y) \wedge T(\phi) \tag{3.14}$$

Moreover, for each free variable x we add a $Sing(X_x)$ formula. The converse translation T' can be defined similarly as written in [6]. □

Example 3.2. *The following example WSkS formula ϕ is in the restricted syntax and is equivalent to the formula from Example 3.1:*

$$\phi \stackrel{def}{=} \exists P : Sing(P) \wedge P \subseteq X \wedge \forall R : Sing(R) \wedge (P = R \vee R \not\subseteq X) \tag{3.15}$$

We will further restrict the syntax of WSkS. A WSkS formula ϕ is in the *prenex normal form* (abbreviated as PNF) if and only if it is of the form

$$\phi = Q_n X_n \dots Q_2 X_2 Q_1 X_1 : \ \psi(\mathbb{X}) \tag{3.16}$$

where $Q_i \in \{\exists, \forall\}$, $X_i \in \mathbb{X}$, $\forall 1 \leq i \leq n$, and ψ is a quantifier-free formula in the restricted syntax as defined above with the additional requirement that negation occurs only on atomic formulae. We denote $Q_n X_n \dots Q_2 X_2 Q_1 X_1$ as the *prefix* of ϕ and $\psi(\mathbb{X})$ as the *matrix* of ϕ.

A WSkS formula ρ is in the *existentially-quantified prenex normal form* (abbreviated as ∃PNF), if and only if its form is

$$\rho = \exists \mathcal{X}_{m+1} \neg\exists \mathcal{X}_m \dots \neg\exists \mathcal{X}_2 \neg\exists \mathcal{X}_1 : \ \psi(\mathbb{X}) \tag{3.17}$$

where $\exists \mathcal{X}_i$, for $\mathcal{X}_i \subseteq \mathbb{X}$, is a (possibly empty) sequence $\exists X_a \dots \exists X_b$ of consecutive existential quantifications and ψ is again a quantifier-free formula in the restricted syntax over $\mathbb{X}$.

Proposition 3.2. *There is a translation from WSkS formulae in the restricted syntax to equivalent formulae in ∃PNF.*

Proof. Let us consider only $\wedge$, $\vee$ and $\neg$ used in formulae as logical operators. This can be achieved by applying the rules $\psi \Leftrightarrow \phi \mapsto (\neg\psi \vee \phi) \wedge (\psi \vee \neg\phi)$ and $\psi \Rightarrow \phi \mapsto \neg\psi \vee \phi$, which preservers logical equivalence.

The formula is first translated to the PNF, moving all quantifiers to the left of the formula, renaming variables if needed. Quantifications with unused variables are removed. We are using the following transformations, where $Q \in \{\exists, \forall\}$ and $\circ \in \{\vee, \wedge\}$:

$$\neg(\exists x\phi) \equiv \forall x\neg\phi \tag{3.18}$$
$$\neg(\forall x\phi) \equiv \exists x\neg\phi \tag{3.19}$$
$$Qx\phi \circ \psi \equiv Qx(\phi \circ \psi) \tag{3.20}$$
$$\phi \circ Qx\psi \equiv Qx(\phi \circ \psi) \tag{3.21}$$

Note that there can be no free occurrences of a quantified variable x in ψ and ϕ in formulae 3.20 and 3.21 respectively.

All universal quantifiers are transformed to existential quantifiers according to the equivalence $\forall x.\ \phi \equiv \neg\exists x.\ \neg\phi$. Lastly, all negations in the matrix are shifted to the atoms according to the following laws:

$$\neg\neg\phi \equiv \phi \tag{3.22}$$
$$\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi \tag{3.23}$$
$$\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi \tag{3.24}$$

The resulting formula is in the existentially-quantified prenex normal form. □

Example 3.3. *The following example WSkS formula ψ is in the existentially-quantified prenex normal form and is equivalent to the formula from Example 3.1:*

$$\psi \stackrel{def}{=} \exists P : \forall R : Sing(P) \wedge P \subseteq X \wedge Sing(R) \wedge (P = R \vee R \not\subseteq X) \tag{3.25}$$

3.4 Deciding WSkS

A decision procedure is an algorithm that given a formula ϕ returns VALID iff ϕ is valid, SAT iff ϕ is invalid but satisfiable (additionally yielding two assignments $\mathcal{M}_{true}$ and $\mathcal{M}_{false}$ such that $\mathcal{M}_{true} \models \phi$ and $\mathcal{M}_{false} \not\models \phi$) and UNSAT iff ϕ is unsatisfiable.

Some properties of verified systems can be checked by being modeled by a set of WSkS formulae and then deciding if the given formulae hold for all variable assignments, therefore the checked system being valid, or either look for a non-satisfying assignment which leads to a violation of the specified properties..

The presented decision procedure for WSkS makes use of the close link between WSkS formulae and automata, i.e. a given formula is transformed to a corresponding finite tree automaton and its language is further examined.

The contents of this section is based on the exposition in [6].

Definition 3.1. *A set L of tuples of finite sets of words is* definable in WSkS *if there is a formula ψ of WSkS with free variables $X_1, \ldots, X_n$ such that*

$$(S_1, \ldots, S_n) \in L \text{ if and only if } S_1, \ldots, S_n \models \psi. \tag{3.26}$$

Each tuple of finite sets of words $S_1, \ldots, S_n \subseteq \{1, \ldots, k\}^*$ is then identified to a tree $(S_1, \ldots, S_n)^\sim$ over the alphabet $\{0, 1, \bot\}^n$ where any string containing 0 or 1 is k-ary and $\bot^n$ is a constant symbol, such that

$$dom((S_1, \ldots, S_n)^\sim) \stackrel{def}{=} \{\epsilon\} \cup \left\{ pi \;\middle|\; \exists p' \in \bigcup_{j=1}^{n} S_j.p \leq p', 1 \leq i \leq k \right\}. \tag{3.27}$$

The symbol at the position p:

$$(S_1, \ldots, S_n)^\sim(p) = \alpha_1 \ldots \alpha_n \tag{3.28}$$

is defined as follows:

- $\alpha_i = 1$ iff $p \in S_i$,
- $\alpha_i = 0$ iff $p \notin S_i \wedge \exists p' : p \cdot p' \in S_i$,
- $\alpha_i = \bot$ otherwise.

Lemma 3.1. *If a set of tuples of finite subsets of $\{1, \ldots, k\}^*$ is definable in WSkS, then $\widetilde{L} \stackrel{def}{=} \{(S_1, \ldots, S_n)^\sim \mid (S_1, \ldots, S_n) \in L\}$ is in Rec.*

Proof. We have shown in Section 3.3 that every formula in WSkS can be translated into an equivalent formula in the restricted syntax. We can now prove the lemma by induction on the structure of the formula ψ which defines the language L.

Let us assume that all variables in ψ are bound at most once in the formula and also that there is a fixed total ordering $<$ on the variables.

If ϕ is a subformula of ψ with free variables, w.r.t. some ordering, $Y_1 < \cdots < Y_n$, we construct the automaton $\mathcal{A}_\phi$ over the alphabet $\{0, 1, \bot\}^n$ such that $(S_1, \ldots, S_n) \models_2 \phi$ if and only if $(S_1, \ldots, S_n)^\sim \in L(\mathcal{A}_\phi)$. □

In the following we limit ourselves to the case when $k = 2$; an extension to an arbitrary k is straightforward. As the induction base, for each atomic subformula of ψ we construct the automaton $\mathcal{A}_\psi$ according to the following rules:

- The automaton $\mathcal{A}_{Sing(X)} = (\{q, q'\}, \{0, 1, \bot\}, \delta, \{q'\})$ where δ is defined as follows:

$$\begin{aligned} &\xrightarrow{\bot} && q \\ (q, q) &\xrightarrow{1} && q' \\ (q, q') &\xrightarrow{0} && q' \\ (q', q) &\xrightarrow{0} && q' \end{aligned}$$

- The automaton $\mathcal{A}_{X\subseteq Y} = (\{q\}, \{0,1,\bot\}^2, \delta, \{q\})$, with $X < Y$, where δ is defined as follows:

$$\begin{array}{rcl} & \xrightarrow{\bot\bot} & q \\ (q,q) & \xrightarrow{00} & q \\ (q,q) & \xrightarrow{\bot 0} & q \\ (q,q) & \xrightarrow{01} & q \\ (q,q) & \xrightarrow{11} & q \\ (q,q) & \xrightarrow{\bot 1} & q \end{array}$$

- The automaton $\mathcal{A}_{X=Y1} = (\{q, q', q''\}, \{0,1,\bot\}^2, \delta, \{q''\})$, where δ is defined as follows:

$$\begin{array}{rcl} & \xrightarrow{\bot\bot} & q \\ (q,q) & \xrightarrow{1\bot} & q' \\ (q,q'') & \xrightarrow{00} & q'' \\ (q',q) & \xrightarrow{01} & q'' \\ (q'',q) & \xrightarrow{00} & q'' \end{array}$$

  Note that the automaton for $X = Y2$ is obtained similarly by changing $(q', q) \xrightarrow{01} q''$ to $(q, q') \xrightarrow{01} q''$.

- The automaton $\mathcal{A}_{X=\epsilon} = (\{q, q'\}, \{0,1,\bot\}, \{q'\}, \delta)$, where δ is defined as follows:

$$\begin{array}{rcl} & \xrightarrow{\bot} & q \\ (q,q) & \xrightarrow{1} & q' \end{array}$$

Now as the induction step, we will consider only logical operations $\vee$, $\wedge$, $\neg$ and $\exists$ (as defined in the restricted syntax):

- $\psi = \psi_1 \vee \psi_2$ — let $\overline{X}_i$ be the set of free variables of ψ_i respectively, and $\overline{X}_1 \cup \overline{X}_2 = \{Y_1, \ldots, Y_n\}$ with some ordering $Y_1 < \ldots < Y_n$. Then we successively apply the i-th cylindrification to the automaton of ψ_1 (resp. ψ_2) for the variables Y_i which are not free in ψ_1 (resp. ψ_2), so the automata for ψ_1 and ψ_2 can recognize the solutions of ψ_1 and ψ_2, with free variables $\overline{X}_1 \cup \overline{X}_2$. Then the automaton $\mathcal{A}_\psi$ is obtained as the union of these automata.

- $\psi = \psi_1 \wedge \psi_2$ — is obtained similarly as the automaton for the connective $\vee$ by performing the *intersection* of automata.

- $\psi = \neg\psi_1$ — then $\mathcal{A}_\psi$ is the automaton accepting the complement of $L(\mathcal{A}_{\psi_1})$.

- $\psi = \exists X.\psi_1$ — assuming that X corresponds to the i-th component, then $\mathcal{A}_\psi$ is the i-th projection of $\mathcal{A}_{\psi_1}$.

Example 3.4. *Consider the following WSkS formula:*

$$\rho \stackrel{def}{=} \exists P : Sing(P) \wedge P \not\subseteq X. \tag{3.29}$$

Deciding validity of ρ consists of constructing the automaton $\mathcal{A}_\rho$ corresponding to the formula ρ and then checking the language $L(\mathcal{A}_\rho)$ for universality, i.e. if any non-accepting state is reachable. As depicted in Figure 3.1 we first construct automata $\mathcal{A}_{Sing(P)}$ and $\mathcal{A}_{P\not\subseteq X}$ for subformulae $Sing(P)$ and $P \not\subseteq X$ respectively. Then we construct the product automaton of $\mathcal{A}_{Sing(P)}$ and $\mathcal{A}_{P\not\subseteq X}$ yielding the automaton $\mathcal{A}_{Sing(P)\wedge P\not\subseteq X}$. Finally we make a projection of the automaton by erasing the track corresponding to the quantified variable P and obtain the resulting automaton $\mathcal{A}_\rho$.

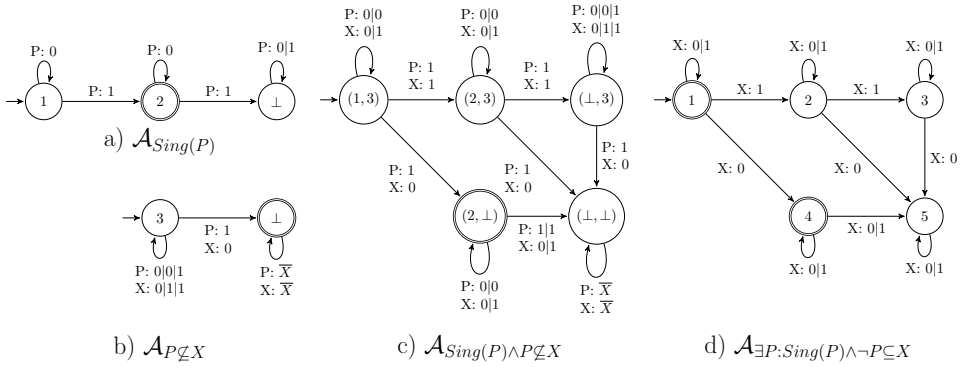


Figure 3.1: Finite automata corresponding to all the subformulae of the given formula $\rho \stackrel{def}{=} \exists P : Sing(P) \wedge P \not\subseteq X$

In implementation terms existential quantification is therefore equivalent to the operation of a projection on the automaton. This raises an issue with final states of the constructed automaton $\mathcal{A}$, since the projection erases some of the transition tracks of the automaton and thus may introduce non-determinism. Such an automaton has to be determinised, yielding DTA $\mathcal{A}'$.

However, the automaton $\mathcal{A}'$ is not enough, since searched witness of some model can be longer after the operation of projection. $\mathcal{A}'$ needs to be further modified to correctly represent encodings of all models of formula. Consider e.g. the assigment $X \mapsto \{111\}, Y \mapsto \{11\}$, encoded as $\binom{X:001}{Y:010}$, as being a model of formula ψ. When we perform projection on X over the model, the resulting encoding $(Y : 010)$ is not a valid encoding of the model $Y \mapsto \{11\}$, which would be $(Y : 01)$.

Definition 3.2. *We define the* right-quotient *of a language L with a language L' as:*

$$L/L' = \{w \mid \exists u \in L' : w \cdot u \in L\}. \tag{3.30}$$

Now we define the language L^i as

$$L^i = \{w \in \{\{0,1\}^k\}^* \mid \text{the } j\text{-th track of word } w \text{ is of the form } 0^* \text{ for } j \neq i\}. \tag{3.31}$$

The language of the automaton $\mathcal{A}'$ after projection γ_i on the i-th track is then defined as

$$L(\mathcal{A}') = \gamma_i(L(\mathcal{A})/L^i). \tag{3.32}$$

In MONA [1] this is done in $\mathcal{O}(n)$ time by using the breadth-first search by backwards exploration of the automaton from final states followed by the subset construction of the determinised automaton yielding only reachable states.

Proposition 3.3. *Since there is a* one-to-one *correspondence between models of ϕ and trees accepted by $\mathcal{A}_\phi$, ϕ is satisfiable iff $L(\mathcal{A}_\phi)$ is non-empty and valid iff $L(\mathcal{A}_\phi)$ is universal.*

The computational complexity of WSkS is in the NONELEMENTARY class [20], so given a Turing machine $\mathcal{M}$ deciding a satisfiability of WSkS formulae, for any $u \geq 0$, there are infinitely many n for which a computation of $\mathcal{M}$ for some sentence of length n requires at least

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}_{u}$$

steps.

Chapter 4

MONA

MONA [1] is one of the early implementations of a decision procedure for the WSkS logic, namely for $k = 1$ and $k = 2$ (note that it can be shown that an arbitrary k can be transformed into formula in WS2S). WS1S can be used for a description of linear structures like linked lists or chains, while WS2S is mainly used for binary tree structures like a binary tree or a binary heap. After many years it is still the best and fastest approach for deciding WSkS formulae with the use of deterministic automata. It is an implementation of the decision procedure from Chapter 3.4 with a few tweaks that we describe in this chapter.

The development of MONA started in 1984 and in the following years numerous approaches were tried out and a number of fine optimizations were discovered. In this chapter we review some of the design choices and implementation tricks that stand behind its success.

4.1 The main issue of the use of deterministic tree automata

Considering a WSkS formula with a fixed number of quantifier alternations N, the decision method outlined in the previous section works in the time which is a tower of exponentials with the height being $O(N)$.

This is mainly because every time we encounter a sequence of quantifiers, we have to do a projection, which yields a non-deterministic automaton, even if the input automaton was deterministic. When we encounter a negation of a formula, we have to use determinization in order to complement the automaton, which requires in general an exponential time and the space w.r.t the number of states of the automaton.

Therefore every time non-determinism is introduced to the automaton, the automaton is determinised and the information about the original states is forgotten. So the MONA approach has issues with extensive the use of an automaton complementation and since there is no known tree automaton complementation technique better than bottom-up determinization of the automaton followed by swapping final and non-final states, MONA had to come up with heavy optimizations and heuristics to achieve good results.

4.2 The used optimizations

4.2.1 Using BDDs for automata representation

BDDs were introduced to solve problems of large input alphabets, which also allowed numerous specialized algorithms to be used.

BDDs were described in Section 2.4; they are useful for its compactness, canonicity and efficient manipulation. MONA uses shared MTBDDs with roots and leaves representing states of an automaton. The use of BDDs for representation of transition relation proved to have the highest effect on the formulae that could not be decided in a fixed limit of time that was set up during benchmarks.

4.2.2 Caching

The implementation of the BDDs, as stated in Section 4.2.1, is optimized to minimize the number of cache misses that occur, since it was discovered that cache misses dominate the running time for both the unary and binary BDD apply operations on a 296 MHz UltraSPARC CPU with 1 GB RAM.

Nodes are thus stored directly under the hash address to minimize the number of cache misses, as opposed to the traditional approach that stores nodes separately from the hash table containing pointers to them, which roughly doubles the time to access a node.

4.2.3 Eager minimization

Whenever MONA performs the product or projection operation during the translation from a formula to an automaton, the Myhill-Nerode minimization takes place, since it is preferable to operate with as small automata as possible. However, this approach was shown to be excessive, since the minimization procedure often exceeds half of the total running time.

Alternatives were introduced — using one final minimization, minimizing only after projection or minimizing only after product, which had different effects and were dependent on particular benchmarks.

4.2.4 Guided tree automata

The set of states is partitioned in order to split a large tree automaton into several smaller ones to address expensive computations caused by three-dimensional transition tables. This however requires for user to specify the *guide* which is a top-down deterministic tree automaton that assigns state spaces to the nodes of a tree.

4.2.5 Directed Acyclic Graph representation

The frontend of the MONA tool is parsing the input files with specification of WSkS formulae. This file is converted to the inner representation of automata-theoretic operations, that are further translated to the resulting automaton.

There are, however, many common subformulae with a similar structure, especially if we talk about *signature equivalence*, which holds for two formulae ϕ and ψ if there is an order-preserving renaming of the variables in formulae such that the representations of ϕ and ψ become identical.

It holds for the BDD representation that automata for signature-equivalent trees are isomorphic in the sense that only node labels differ, which means these representations can be reused simply by renaming variable nodes. Thus MONA represents input formulae in the form of *directed acyclic graph* and not a tree.

4.2.6 Three-valued logic and automata

Since formulae are translated to the restricted syntax that uses only second-order variables, first-order variables are encoded as singletons. This however raises the issue of *restrictions*, i.e. a formula ϕ holds only when some external associated restrictions hold. Since a restriction is also a formula, the main issue is that ϕ is now undefined outside the restriction. Note that for a first-order variable, the restriction is that it is a singleton set.

The nature of these problems are solved by using a three-valued logic. So for a restricted subformula ϕ we associate a restriction ϕ_R. And if for some valuation ϕ_R does not hold, then the formulae containing ϕ are assigned the third value *don't-care*.

A special operation converts the rejecting states to don't-cares for the restriction formulae and other automaton operations are modified so these nonacceptance of restrictions are propagated properly.

4.2.7 Formula reductions

Various optimizations of formulae takes place in the DAG specified in Section 4.2.5 before the final translation to the automaton. Reductions are based on the syntactic analysis that tries to identify valid subformulae and equivalences among them.

MONA performs few kinds of reductions:

1. Simple equality and Boolean reductions that can be described by simple rewrite rules like $\phi \wedge \phi \mapsto \phi$, etc. These rewrite steps guarantee a reduction of complexity, but will not cause significant improvements in the running time, since they rarely apply in realistic situations. However, they are cheap and may yield small improvements.

2. Special quantifier reductions. The basic idea is to apply a rewrite step which removes quantifiers where they are not useful, as following:

$$\exists X.\phi \mapsto \phi[T/X] \tag{4.1}$$

   provided that $\phi \Rightarrow X = T$ is valid formula and the term T is satisfying that $\mathit{freeVars}(T) \subseteq \mathit{freeVars}(\phi)$, where $\mathit{freeVars}(T)$ denotes the set of free variables in some subformula T. This is further restricted to a different rewrite rule:

$$\exists X_i.\phi \mapsto \phi[X_j/X_i] \tag{4.2}$$

   provided that $\phi \equiv \ldots \wedge X_i = X_j \wedge \ldots$ and X_j is some variable other than X_i. This, however, is not guaranteed to yield better results.

Chapter 5

Deciding WSkS with Non-deterministic Automata

The MONA tool implementation of the decision procedure for WSkS from Section 3.4 uses deterministic bottom-up tree automata (as described in Chapter 4) and so every time non-determinism might be introduced, such as through the union and the projection corresponding to the disjunction and the existential quantification respectively, the automaton is determinised using the subset construction and the information about the original states is forgotten.

While this approach makes automaton complementation easy (since a complete deterministic automaton require only swapping final with non-final states), the extensive determinisation is still a serious drawback. The MONA tool is thus heavily optimized for the use of deterministic finite (tree) automata, as we introduced in Section 4.2. However, a decision procedure that uses non-deterministic automata needs to deal with the issue of complementation, for which there is currently no known algorithm that avoids determinisation and the respective state explosion.

In practice, the representation of all models of ϕ is not always necessary and any model or an invalid assignment to free variables suffices, therefore constructing the full automaton $\mathcal{A}_\phi$ representing all models of ϕ can be avoided.

Current trends in formal verification and theory of automata tend to use non-deterministic automata instead of deterministic ones. This is due to the fact that there exist optimized libraries for their use, like VATA [18], and recently new algorithms that are efficient in practice were developed even for time complex operations like language inclusion or universality testing, which are PSPACE-complete for finite word automata and EXPTIME-complete for tree automata.

Here we propose to use an algorithm based on the principle of antichains [4], defined in the following sections, and search for an accepting or a non-accepting state *on-the-fly* without constructing the automaton corresponding to the given formula in the first place.

5.1 Antichain-based universality testing of NFAs

We will give a brief introduction to the universality testing for non-deterministic finite word automata by an algorithm that uses a combination of the simulation-based and the antichain-based approaches [4].

Example 5.1. *Consider testing the validity of the formula ρ from Example 3.4:*

$$\rho \stackrel{def}{=} \exists P : Sing(P) \wedge P \not\subseteq X. \tag{5.1}$$

Given the automaton $\mathcal{A}_\rho$ corresponding to the formula ρ (see Section 3.4), testing validity of ρ is equivalent to testing universality of the language $L(\mathcal{A}_\rho)$, which can be done by searching for an accepting state in the complemented automaton.

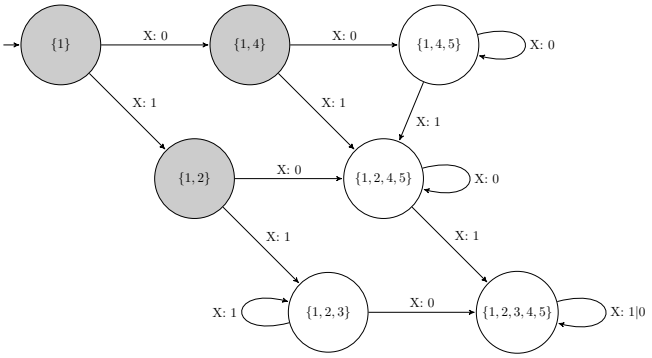


Figure 5.1: Comparison of the antichain-based (grey nodes) and the classical (all nodes) approach to universality checking.

When using the textbook approach for testing universality (by using the subset construction and complementation), 7 states are generated as depicted in Figure 5.1 If we use the antichain-based approach, then macro-states $\{1, 2\}$ and $\{1, 4\}$ are simulated by the initial state $\{1\}$ so we can discard these states. Since we have not reached an accepting state, we can conclude that the language of the complemented automaton $\mathcal{A}_{\neg\rho}$ is empty, then $\mathcal{A}_\rho$ is universal and ρ is thus valid.

Definition 5.1. *A* simulation *on a finite automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is a relation $\preceq \subseteq Q \times Q$ such that $p \preceq r$ implies the following two conditions (i) $p \in F \Rightarrow r \in F$ and (ii) for every transition $p \xrightarrow{a} p'$, there exists a transition $r \xrightarrow{a} r'$ such that $p' \preceq r'$.*

We use $A^{\subseteq}$ to denote the set of relations over the states of $\mathcal{A}$ that imply language inclusion, that is for all $\leqq \in A^{\subseteq}$ it holds that $p \leqq r \Rightarrow L(p) \subseteq L(r)$. Analogously we define the set of relations over the macro-states $\mathcal{A}$ that imply language inclusion as $\mathcal{A}^{\sqsubseteq}$.

Lemma 5.1. *Given a simulation $\preceq$ on a NFA $\mathcal{A}$, it holds that $\preceq \in \mathcal{A}^{\subseteq}$.*

Proof. A proof can be found in [4]. □

The universality problem for a NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is to decide whether $L(\mathcal{A}) = \Sigma^*$. This problem is hard, it actually is PSPACE-complete, however, the heuristic described in the following works well for many real-world examples.

The naive algorithm performs determinization using the subset construction to obtain the deterministic automaton $\mathcal{A}_D$, then complements it to obtain $\overline{\mathcal{A}_D}$ and finally checks that there is no reachable accepting state in $\overline{\mathcal{A}_D}$ (in case there is, the language is not universal).

The algorithm proposed in [4] runs the subset construction procedure *on-the-fly*, avoiding explicit construction of $\overline{\mathcal{A}_D}$, and checks if any accepting macro-state is reachable. This procedure is further augmented with the following two optimizations.

The first optimization is based on the following lemma.

Lemma 5.2. *Let P, R be two macro-states of a NFA $\mathcal{A}$, and $\preceq$ be a relation from $\mathcal{A}^{\subseteq}$. Then, $P \preceq^{\forall\exists} R$ implies $L_{\mathcal{A}}(P) \subseteq L_{\mathcal{A}}(R)$, where $P \preceq^{\forall\exists} R$ stands for $\forall p \in P. \exists r \in R : p \preceq r$.*

The relation $\preceq$ can be any relation on the states of $\mathcal{A}$ that implies language inclusion, e.g. the maximum simulation or the identity relation. When two states P and R such that $P \preceq^{\forall\exists} R$ are encountered during the search for a non-accepting state, we can discard R, because $L(P) \subseteq L(R)$ so P has a higher chance to find a non-accepting state. The other optimization is based on the observation that $L_{\mathcal{A}}(P) = L_{\mathcal{A}}(P \setminus \{p_1\})$ if there is some $p_2 \in P \setminus \{p_1\}$ such that $p_1 \preceq p_2$, which is also a simple consequence of Lemma 5.2.

Algorithm 1: Checking reachability of a final state using optimized algorithm [4]

Input: A macro-state I of an automaton $\mathcal{A}$, a relation $\leqq \in A^{\sqsubseteq}$ on the macro-states of $\mathcal{A}$, a successor function δ and a predicate `IsFinal(F)` that decides whether a macro-state F is final

Output: TRUE iff there exists a final state in $\mathcal{A}$ reachable from I, FALSE otherwise

Function `IsFinalReachable`(**state** I, **relation** $\leqq$, **post** δ, **pred** `IsFinal`)

```
1  if IsFinal(I) then
2      return TRUE;
3  Processed := ∅;
4  Next := {I};
5  while Next ≠ ∅ do
6      Pick and remove a macro-state R from Next and move it to Processed;
7      if IsFinal(R) then
8          return TRUE;
9      foreach P ∈ δ(R) do
10         if ¬∃S ∈ Processed ∪ Next s.t. P ≦ S then
11             Remove all S from Processed ∪ Next s.t. S ≦ P;
12             Add P to Next;
13 return FALSE;
```

The algorithm for testing universality of NFA is based on Algorithm 1, which works as follows. While there are macro-states to be processed, and no accepting macro-state has been found, one of the macro-states from *Next* is chosen and moved to the *Processed* set. All successors of the macro-state are generated, minimized and moved to *Next* unless there is already some $\leqq$-bigger macro-state in *Next* or in *Processed*. If a new macro-state is added to *Next*, then all $\leqq$-smaller states are pruned out of both *Next* and *Processed*.

Testing universality of an automaton $\mathcal{A}$ is thus equivalent to the searching for an accepting state in the automaton $\overline{\mathcal{A}_D}$ obtained by determinization and complementation of the set of final states.

Lemma 5.3. *Given the automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, the function `IsFinalReachable`($\{q\}$, $\leqq \in \mathcal{A}^{\sqsubseteq}$, $(\lambda R.\ \{\{s\} \mid \exists r \in R :\ r \xrightarrow{a} s \in \delta\})$, $(\lambda\ R. R \cap F \neq \emptyset)$) from Algorithm 1 returns* TRUE *iff there is a reachable final state from state q in $\mathcal{A}$.*

Proof. A proof can be found in [7]. □

Lemma 5.4. *The relation $\subseteq^{-1}$ is a simulation on the complemented automaton $\overline{\mathcal{A}_D}$.*

Proof. A proof can be found in [7] □

Lemma 5.5. *The language of an automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is universal iff the function* `IsFinalReachable`*(I, $\leqq \in \mathcal{A}^{\sqsubseteq}$, $(\lambda R.\ \{\{s\} \mid \exists r \in R : r \xrightarrow{a} s \in \delta\})$, $(\lambda\ R.R \cap F \neq \emptyset)$) from Algorithm 1 returns* TRUE*.*

5.2 Deciding WS1S with NFAs

In this section we will describe the proposed decision procedure on the simpler case for $k = 1$, i.e. WS1S. In the next section, we will extend the procedure to an arbitrary k.

Given a formula ϕ, the construction of the automaton $\mathcal{A}_\phi$ representing all models of ϕ is not always necessary and in many applications any model or an invalid assignment suffices. Therefore we can exploit the antichain-based techniques defined in the previous section and search for an accepting or a non-accepting state *on-the-fly* while pruning the search space.

First, the formula ϕ is transformed to a logically equivalent formula φ in the existential prenex form (see Section 3.3):

$$\varphi \stackrel{def}{=} \exists \mathcal{X}_{m+1} \neg \mathcal{X}_m \neg \ldots \neg \exists \mathcal{X}_2 \neg \exists \mathcal{X}_1 : \pi.$$

Supposing there are m negations in the prefix of the formula φ, we create a hierarchical family of WSkS formulae $\Phi = \{\varphi_0, \ldots, \varphi_m\}$ where

$$\varphi_0 \stackrel{def}{=} \pi \tag{5.2}$$

and for all $0 \leq i \leq m-1$

$$\varphi_{i+1} \stackrel{def}{=} \neg \exists \mathcal{X}_{i+1} :\ \varphi_i. \tag{5.3}$$

The relation between this family of formulae Φ and the formula φ is depicted as follows:

$$\varphi \stackrel{def}{=} \exists \mathcal{X}_{m+1} \neg \exists \mathcal{X}_m \neg \ldots \neg \exists \mathcal{X}_2 \neg \exists \mathcal{X}_1 : \underbrace{\pi}_{\varphi_0} \quad \underbrace{\qquad\qquad}_{\varphi_1} \quad \underbrace{\qquad\qquad\qquad}_{\varphi_2} \quad \underbrace{\qquad\qquad\qquad\qquad}_{\varphi_m} \tag{5.4}$$

We will use the notation $\Gamma(G)$ to denote the alphabet defined as the set of functions $\Gamma(G) = \{f : G \to \{0, 1, \bot\}\}$, with the special case for $G = \emptyset$ defined as $\Gamma(\emptyset) = \{\#\}$. Elements of this set can be viewed as strings of a fixed length $|G|$ over the alphabet $\{0, 1, \bot\}$ where each position in the strings is associated with exactly one element of G.

Given the symbol $f \in \Gamma(G)$ and a set $H \subseteq G$, the *projection* of f over the set H, written as $\omega_H(f)$, can be defined as the restriction of the function f over the set $G \setminus H$, i.e. $\omega_H(f) = f|_{G \setminus H}$. This can be further extended to sets of symbols. Given a set $E \subseteq \Gamma(G)$, we define the projection over a set $H \subseteq G$, written $\omega_H(E)$, as $\omega_H(E) = \{\omega_H(f) \mid f \in E\}$. For a string $w = a_1 \ldots a_n \in \Gamma(G)^*$ we define the operation as $\omega_H(w) = \omega_H(a_1) \ldots \omega_H(a_n)$.

For a language $L \subseteq \Gamma(G)^*$, the projection of L over H, written as $\omega_H(L)$, is defined as $\omega_H(L) = \{\omega_H(f) \mid f \in L\}$. Note that $\omega_H(L) \subseteq \Gamma(G \setminus H)^*$.

Lemma 5.6. *Given the alphabet* $\Gamma(G)$*,* $H \subseteq G$ *and* $X, Y \subseteq \Gamma(G)^*$ *it holds that*

$$X \subseteq Y \Longrightarrow \omega_H(X) \subseteq \omega_H(Y). \tag{5.5}$$

Proof. For each symbol $z \in \omega_H(X)$ there must exist some $x \in X$ such that $z = \omega_H(x)$. Since $X \subseteq Y$ it must also hold that $x \in Y$, and hence it holds that $\omega_H(x) \in \omega_H(Y)$. Therefore $\omega_H(X) \subseteq \omega_H(Y)$. □

As we described in Section 3, there is an issue with the encoding of all models of formula and using the projection operation on the automaton. If the projection were implemented by only removing parts of symbols corresponding to the variables in the existential quantification, the resulting automaton would not accept some of the valid models of the formula, as described by Example 5.2. MONA (see Section 3.4) solves this by adjusting the final states of the automaton by using the breadth-first search algorithm in the linear time after every projection of an automaton.

Note that in the following we will use the symbol $\overline{0}$ as the substitute for the symbol 0^k for some known k.

Example 5.2. *Consider the formula* $P \not\subseteq X$ *and its corresponding automaton* $\mathcal{A}_{P \not\subseteq X}$ *depicted in Figure 5.2 with one final state. After restricting the tracks of the automaton by removing the variable* P *we get the automaton* $\mathcal{A}_{\exists P: P \not\subseteq X}$*b in Figure 5.2b.*

After the restriction the automaton does not accept e.g. the following word 111*,* $111 \notin L(\mathcal{A}_{\exists P: P \not\subseteq X})$*, even though* $X = \{1, 2, 3\} \models \exists P : P \not\subseteq X$*. However, it does accept another encoding of this set namely* $1110 \in L(\mathcal{A}_{\exists P: P \not\subseteq X})$*, which is not a correct encoding according to the rules set in Chapter 3. Hence we must adjust the final states by extending it by the states that can reach them with words* $\overline{0}^*$*.*

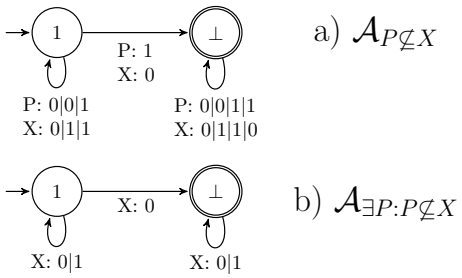


Figure 5.2: The issue with final states after projection is that the language of the resulting automaton restricts encodings of some valid models afterwards.

Adjusting the final states needs a fully constructed automaton, so this cannot be used in an on-the-fly algorithm. Instead, we can either precompute the set of final states inspired by backwards universality testing [12] or use a lazy approach that will determine whether a state is final only when this information is actually needed.

Definition 5.2. *For a set of states* R *of the automaton* $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ *we define the* set of direct predecessors through zero tracks *as the set* $PRE_0(R)$ *of states such that*

$$PRE_0(R) = \{q \in Q \mid \exists r \in R : q \xrightarrow{\overline{0}} r \in \delta\}. \tag{5.6}$$

We denote the reflexive transitive closure of $PRE_0(Q)$ as $PRE_0^*(Q)$. Note that we can define the $POST_0$ function similarly to PRE_0.

Definition 5.3. *Given the automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ we define the* projection $\omega_\mathcal{X}$ *of $\mathcal{A}$ over the set of variables $\mathcal{X}$ as the automaton $\omega_\mathcal{X}(\mathcal{A}) = (Q, \omega_\mathcal{X}(\Sigma), \omega_\mathcal{X}(\delta), I, PRE_0^*(F))$ where*

$$\omega_\mathcal{X}(\delta) = \left\{ p \xrightarrow{\omega(a)} r \mid p \xrightarrow{a} r \in \delta \right\}. \tag{5.7}$$

Note that the result of projection has the final states adjusted according to the PRE_0 relation.

Definition 5.4. *Given the automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ we define the* complementation γ *of $\mathcal{A}$ as the automaton $\gamma(\mathcal{A}) = (2^Q, \Sigma, \delta_\gamma, \{I\}, \{S \subseteq Q \mid S \cap F = \emptyset\})$ where*

$$\delta_\gamma = \{R \xrightarrow{a} S \mid S = \{s \in Q \mid \exists r \in R : r \xrightarrow{a} s \in \delta\}\}. \tag{5.8}$$

Note that this operation γ corresponds to the determinisation of the input automaton using the subset construction followed by swapping final and non-final states of the determinised automaton.

Given a WS1S formula φ in the $\exists$PNF form, the automaton for the matrix π of the formula, $\mathcal{A}_\pi$, can be constructed out of the automata corresponding to the atomic formulae and their complements by using the operations of union and intersection only (see Section 3.4). Further, $\mathcal{A}_\varphi$ could be constructed from the automaton $\mathcal{A}_\pi$ by successive applications of complementation γ and projection ω according to the prefix of φ as these operations were defined before. But we take a different path.

Analogously to the family of formulae Φ, we can define the family of automata $\mathbb{A} = \{\mathcal{A}_0, \ldots, \mathcal{A}_m\}$ as follows:

$$\mathcal{A}_0 = \mathcal{A}_\pi, \tag{5.9}$$

$$\mathcal{A}_{i+1} = \gamma(\omega_{\mathcal{X}_{i+1}}(\mathcal{A}_i)). \tag{5.10}$$

Note that the automaton $\mathcal{A}_i$ then corresponds to the following formula:

$$\varphi_i = \neg\exists\mathcal{X}_i \ldots \neg\exists\mathcal{X}_1.\pi. \tag{5.11}$$

We use the notation $\mathcal{A}_i = (Q_i, \Sigma_i, \delta_i, I_i, F_i)$ to address a particular automaton and its components. There is a further connection between the hierarchical family of formulae Φ and the hierarchical family of automata $\mathbb{A}$ as described by the following lemma.

Lemma 5.7. *For all $0 \leq i \leq m$, the language of $\mathcal{A}_i$ is*

1. universal *iff φ_i is* valid,
2. non-empty *iff φ_i is* satisfiable,
3. non-universal *iff φ_i is* invalid, *and*
4. empty *iff φ_i is* unsatisfiable.

Proof. A proof of this Lemma can be found in [6]. □

Given the automaton $\mathcal{A}_0$ and the sequence of sets of second-order variables $(\mathcal{X}_1, \ldots, \mathcal{X}_m)$ corresponding to the matrix and the prefix of φ respectively, we can classify the formula according to the existence of satisfying and unsatisfying assignments.

Definition 5.5. *We say a formula ϕ is* closed *iff there is no free variable in ϕ, i.e. $freeVars(\phi) = \emptyset$. We further define the* existential closure *of a formula φ as the formula $\exists$-cl such that*

$$\exists\text{-}cl(\varphi) = \exists\mathcal{X} : \ \varphi \tag{5.12}$$

where

$$\mathcal{X} = freeVars(\varphi). \tag{5.13}$$

Note that we can construct an automaton corresponding to the existential closure of a formula φ as $\widetilde{\mathcal{A}_\varphi} = \omega_\mathcal{X}(\mathcal{A}_\varphi)$. It is obvious that since the formula has no free variables, the symbols on transitions in $\widetilde{\mathcal{A}_\varphi}$ are #.

Lemma 5.8. *Let φ be a closed WS1S formula. Then φ has a model iff there is an initial state of $\mathcal{A}_\varphi = (Q, \Sigma, \delta, I, F)$ that is final, i.e.*

$$I_m \cap F_m \neq \emptyset \Leftrightarrow \models \varphi. \tag{5.14}$$

Proof. This is implied by Lemma 5.7 □

We can extend this notion to any WS1S formula ψ by computing the closure of the formula ψ to yield a formula φ and test the validity of the formula as stated by Lemma 5.8.

To optimize the algorithm of validity testing for WS1S formulae, we are going to define the necessary relations so that we can prune the state space during the search for an accepting or a rejecting state.

Definition 5.6. *For all $0 \leq i \leq m$ we define the relation $\leq_i \subseteq Q_i \times Q_i$ as follows:*

$$\leq_0 \ = \ id, \tag{5.15}$$

$$A \leq_{i+1} B \ \Leftrightarrow \ \forall b \in B \exists a \ \in A : a \geq_i b. \tag{5.16}$$

Lemma 5.9. *For all $0 \leq i \leq m$ the relation $\leq_i$ is a simulation on $\mathcal{A}_i = (Q_i, \Sigma, \delta_i, I_i, F_i)$.*

Proof. We will prove this lemma by induction on the level i of the relation.

1. The base case $i = 0$: From the definition of relation $\leq_i$, $\leq_0$ is equal to the identity relation, which is a simulation [4].

2. Now let us take the following induction hypothesis:

$$\leq_n \ \texttt{is a simulation on } \mathcal{A}_n. \tag{5.17}$$

Further we wil prove the lemma for $n + 1$, i.e. that the relation $\leq_{n+1}$ such that

$$A \leq_{n+1} B \overset{def}{\Leftrightarrow} \forall b \in B \exists a \in A . a \geq_n b \tag{5.18}$$

is a simulation.

The relation $\leq_n$ is a simulation on $\omega_{n+1}(\mathcal{A}_n)$, since if $a \geq_n b$ then it also holds that $\omega_{n+1}(a) \geq_n \omega_{n+1}(b)$.

Now to prove that $A \leq_{n+1} B$ is a simulation we need to show that the following formula is true for all $x \in \Sigma$:

$$(A \xrightarrow{x} A' \Rightarrow (\exists B' . B \xrightarrow{x} B' \wedge A' \leq_{n+1} B')) \wedge (A \in F_{n+1} \Rightarrow B \in F_{n+1}). \tag{5.19}$$

We consider the following case split:

a) $(A \xrightarrow{x} A' \Rightarrow (\exists B'.B \xrightarrow{x} B' \wedge A' \leq_{n+1} B'))$:

This condition can be further elaborated into the following formula:

$$\begin{aligned}&(A \xrightarrow{x} \{a' \mid \exists a \in A.a \xrightarrow{x} a' \in \delta_n\}) \Rightarrow \\ &\Rightarrow ((B \xrightarrow{x} \{b' \mid \exists b \in B.b \xrightarrow{x} b' \in \delta_n\}) \wedge \forall b' \in B' \exists a' \in A'.a' \geq_n b')\end{aligned} \tag{5.20}$$

Since $A \leq_{n+1} B$, so $\forall b \in B.\ \exists a \in A.\ b \leq_n a$, it holds that $\forall b' \in \{b' \mid \exists b \in B.b \xrightarrow{x} b' \in \delta_n\}$ there is some $a' \in \{a' \mid \exists a \in A.\ a \xrightarrow{x} a' \in \delta_n\}$ such that $b' \leq_n a'$. Because, if there did not exists such a', then $A \leq_n B$ could not be a simulation in the first place.

b) $A \in F_{n+1} \Rightarrow B \in F_{n+1}$:

A macro-state A is final on the level i if it does not contain a state on the level $i-1$ which is final, i.e. $A \in F_{n+1} \overset{def}{\Leftrightarrow} \neg\exists a \in A.\ a \in F_n$. If there is no final state in A, then there will surely be no final state in B because A has a bigger language than B.

We have shown that both formulae of the conjuction a) and b) are true and thus the relation $\leq_{n+1}$ is a simulation on $\mathcal{A}_{n+1}$.

□

Definition 5.7. *Given the automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ we define the restriction of $\mathcal{A}$ to zero tracks as the automaton $\mathcal{A}^0 = (Q^0, \{\overline{0}\}, \delta^0, I, PRE_0^*(F))$ where*

$$Q^0 = \{q \in Q \mid \exists w \in \overline{0}^* : f \xrightarrow{w} q, f \in I\}, \tag{5.21}$$

$$\delta^0 = \{p \xrightarrow{\overline{0}} q \mid p \xrightarrow{\overline{0}} q \in \delta\}. \tag{5.22}$$

Note that all states are final due to the adjustment of states after projection. The final Algorithm 2 is based on the universality checking algorithm as described in this chapter:

Algorithm 2: Algorithm for deciding the validity of a WSkS formula φ

Input: The initial state I_m of the automaton $A_{\neg\varphi_m}$, a relation on macro-states $\leqq \in \mathcal{A}^{\sqsubseteq}$

Output: TRUE iff the formula φ is valid, FALSE otherwise

```
1 return IsStateAccepting(I_m, m)
  Function IsStateAccepting(state P, level i)
2   if i = 0 then
3     return P ∈ F_0;
4   else
5     foreach p ∈ P do
6       if IsFinalReachable(p, ≦, δ^0_{i-1}, (λ q. IsStateAccepting(q, i − 1)))
        then
7         return FALSE
8     return TRUE
```

5.3 Deciding WSkS

In the previous section we introduced the concept of the algorithm for deciding WS1S formulae using non-deterministic finite automata (or unary tree automata). We will briefly describe an extension of this procedure to an arbitrary k.

We extent the notion of projection to trees. Given a tree $t : \mathbb{N}^* \to \Gamma(G)$ the projection of t over $H \subseteq G$ is defined as $\omega_H(t) = \{(n, \omega_H(f)) \mid (n, f) \in t\}$. Note that $\omega_H(t) : \mathbb{N}^* \to \Gamma(G \setminus H)$.

Similarly to the logic WS1S we will define the hierarchical family of tree automata $\mathbb{A} = \{\mathcal{A}_0, \ldots, \mathcal{A}_m\}$ where

$$\mathcal{A}_0 = (Q_0, \Sigma_0, \delta_0, F_0) \tag{5.23}$$

is a non-deterministic finite tree automaton corresponding to the WSkS formula $\varphi_0 \stackrel{def}{=} \pi(\mathbb{X})$, such that $\Sigma_0 = \Gamma(\mathbb{X})$ and

$$\mathcal{A}_{i+1} = (Q_{i+1}, \Sigma_{i+1}, \delta_{i+1}, F_{i+1}) \tag{5.24}$$

is a tree automaton corresponding to the formula $\varphi_{i+1} \stackrel{def}{=} \neg\exists\mathcal{X}_{i+1} : \varphi_i$ where

$$\begin{aligned} Q_{i+1} &= 2^{Q_i}, && (5.25)\\ \Sigma_{i+1} &= \omega_{\mathcal{X}_{i+1}}(\Sigma_i), && (5.26)\\ F_{i+1} &= \{R \in Q_{i+1} \mid R \cap F_i = \emptyset\}, && (5.27)\\ \delta_{i+1} &= \{(R_1, \ldots, R_t) \stackrel{\omega_{\mathcal{X}_{i+1}}(f)}{\longrightarrow} S\}, && (5.28) \end{aligned}$$

where

$$S = \{s \in Q_i \mid \exists r_1 \in R_1, \ldots, r_t \in R_t.(r_1, \ldots, r_t) \stackrel{f}{\longrightarrow}, s\}. \tag{5.29}$$

In order to be able to talk about possible *futures* of states of automata from family of $\mathbb{A}$ we exploit the notion of languages of the so called *open trees* as defined in [4].

Consider a ranked alphabet Σ, we define a special symbol $\Box \notin \Sigma$ with rank 0, called a *hole*. Then an *open tree* over Σ is a tree over $\Sigma \cup \{\Box\}$ such that all its leaves are labeled by symbol $\Box$. we use $T_\Sigma^\Box$ to denote the set of all open trees over Σ. Given states $q_1, \ldots, q_n$ of the automaton $\mathcal{A} = (Q, \Sigma, \delta, F)$ and an open tree t with leaves $v_1, \ldots, v_n$, a run π of $\mathcal{A}$ on t from $(q_1, \ldots, q_n)$ is defined in similar way as the run on a tree except that for each leaf $v_i, 1 \leq i \leq n$, we have $\pi(v_i) = q_i$. We use $t(q_1, \ldots, q_n) \stackrel{\pi}{\Longrightarrow} q$ to denote that π is a run of $\mathcal{A}$ on t from $(q_1, \ldots, q_n)$ such that $\pi(\epsilon) = q$. We define the notation $t(q_1, \ldots, q_n) \Longrightarrow q$ similarly to runs on trees.

The language of $\mathcal{A}$ accepted from a tuple $(q_1, \ldots, q_n)$ of states of Q is $L_\mathcal{A}^\Box(q_1, \ldots, q_n) = \{t \in T_\Sigma^\Box \mid t(q_1, \ldots, q_n) \Longrightarrow q \text{for some } q \in F\}$. Then language of $\mathcal{A}$ accepted from a tuple $(S_1, \ldots, S_n)$ of sets of states from $\mathcal{A}$ is defined as follows:

$$L_\mathcal{A}^\Box(S_1, \ldots, S_n) = \bigcup_{(s_1,\ldots,s_n) \in S_1 \times \ldots \times S_n} L_\mathcal{A}^\Box(s_1, \ldots, s_n). \tag{5.30}$$

Given the vector $\vec{q_n} = (q_1, \ldots, q_n)$ of states, we use the notation $L_\mathcal{A}^\Box$ to denote the language of open trees $L_\Sigma^\Box$. Further we use the notation $\vec{q_n}[e \mapsto s]$, where $1 \leq e \leq n$, to denote the vector $(q_1, \ldots, q_{e-1}, s, q_{e+1}, \ldots, q_n)$. This notion can be further extended to vectors of sets of states.

Further we extent the notion of pruning states as defined in previous sections over the languages of open trees by the following lemmas.

Lemma 5.10. *Let $\sqsubseteq_i \subseteq Q_i \times Q_i$, $0 \leq i \leq m-1$, be a relation on the states of the automaton $\mathcal{A}_i$ such that implies inclusion of languages of open trees, i.e.*

$$a \sqsubseteq_i b \Rightarrow \forall 1 \leq e \leq n. \forall \vec{q_n} \in Q_i^n . L_{\mathcal{A}_i}(\vec{q_n}[e \mapsto a]) \subseteq L_{\mathcal{A}_i}(\vec{q_n}[e \mapsto b]) \tag{5.31}$$

then $\forall p, r \in Q_i$, $S \subseteq Q_i$ and $\vec{V_n} \subseteq Q_i^n$, $1 \leq e \leq n$, it holds that

$$p \sqsubseteq_i r \Rightarrow L_{\mathcal{A}_{i+1}}(\vec{V_n}[e \mapsto (\{p, r\} \cup S)]) \subseteq L_{\mathcal{A}_{i+1}}(\vec{V_n}[e \mapsto (\{r\} \cup S)]) \tag{5.32}$$

where $(\{p, r, \} \cup S)$, $(\{r\} \cup S)$ are states of the automaton $\mathcal{A}_{i+1}$.

Proof. Proof can be found in [4]. □

Definition 5.8. *For all $0 \leq i \leq m$ we define the family of relations $\{\sqsubseteq_i^0, \ldots, \sqsubseteq_i^i\}$ over the states of the automaton $\mathcal{A}_i$ such that $\forall 0 \leq k \leq i : \sqsubseteq_i^k \subseteq Q_i \times Q_i$ as follows:*

$$\sqsubseteq_0 = id, \tag{5.33}$$

$$\sqsubseteq_i = \{(A, B) \mid \forall b \in B \exists a \in A. a \sqsupseteq b\}. \tag{5.34}$$

We can use the Algorithm 2 described in previous section to decide any WSkS with operation of projection extended over the trees. Further we can use relations previously defined to prune the state space similarly to word automata.

Chapter 6

Implementation

As depicted in Figure 6.1, the input of the implemented application is a collection of WS1S or WS2S formulae written using the MONA syntax as described in Appendix B. The frontend of the created application is reused and slightly modified parser of tool MONA that is based on `yacc` [13] parser generator. The input file is parsed into an intermediate representation as an Abstract Syntax Tree (AST) with logical connectives or atomic formulae as tree nodes. While parsing the input file, a symbol table is filled up with the used first or second-order variables and defined predicates or macros.

We enhance the frontend as follows. Mapping of variables to MTBDD tracks can be constructed in several possible ways. Either it can correspond to the place of the definition of the variable during the parsing process, chosen randomly or match the sequence of quantifiers from the prefix. We prefer the last option, because projection of lower-leveled MTBDD nodes is more efficient than higher-level nodes.

The AST is then flattened according to the rules for translation to the restricted syntax and transformed into the existential prenex normal form (see Section 3.3). The resulting AST is then broken into the matrix (a quantifier free formula) and the prefix (a sequence of quantifiers).

The matrix of the formula is converted to the NTA with the use of the `libvata` library [17]. Along with the prefix of the formula this is an input for the decision procedure which decides whether the formula is either (*i*) *valid*, (*ii*) *invalid, but satisfiable*, or (*iii*) *unsatisfiable*.

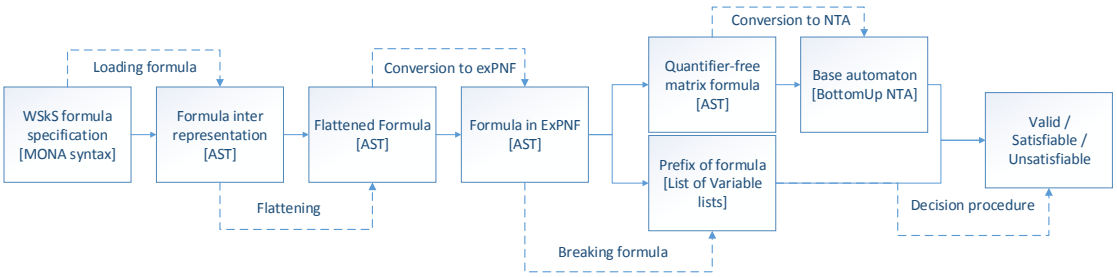


Figure 6.1: Transformation of data through the decision procedure

6.1 Manipulation with NTA

For the representation of automata used in the decision procedure, the `libvata` library [17], written in C++, was chosen for being an efficient and open source library that exploits some of the recent developments of algorithms for NTA manipulation. Its main use is in the fields of formal verification, but it can be efficiently used in other domains as well.

`libvata` supports two possible encodings of tree automata (*explicit* and *semi-symbolic*) which differ in the way they store the transition relations of automata. The semi-symbolic representation uses MTBDDs for storing the transitions of automata as described in Section 2.3. This is mostly intended for automata with large input alphabets like in the case of some decision procedures of logics such as WSkS [1] or MSO [22] (monadic second-order logic). The library is designed in a modular way, so it can be easily extended with new encodings and operations.

The library provides a command line interface, so the input of the library is usually a text representation in Timbuk format [2], which is parsed and transformed into inner representation with states, transition, alphabets, etc. However, we can also build the automaton from scratch using API functions for appropriate encodings. Each automaton can be serialized back to an output format (like Timbuk) through serializers.

We are going to use the semi-symbolic encoding that provides both bottom-up and top-down representation of tree automata. These representation differs in storage of MTBDD where for top-down representation the storage is more complex, while on the other hand, in the bottom-up representation arity can be inferred from the arity of the tuples on the left-hand side of the transition. As for the representation of automata, we have chosen the bottom-up representation.

6.2 Decision procedure

Algorithm 3: Implementation of deciding validity of WSkS formula

Input: A state I_m of the automaton A_{φ_m}, a level of determinization m
Output: TRUE iff φ_m is valid, FALSE otherwise
Data: `StateCache` caches the answers of the `StateIsFinal` function

```
1  return IsStateFinal(I_m, m);

   Function IsStateFinal(state P, level i)
2      isFinal := StateCache[P, m];
3      if isFinal ≠ ⊥ then
4          return isFinal;
5      if m = 0 then
6          return Q ∈ F_0;
7      else
8          foreach q ∈ Q do
9              if CheckForAcceptingState(q, m − 1) then
10                 StateCache[P, m] := FALSE;
11                 return FALSE;
12         StateCache[P, m] := TRUE;
13         return TRUE;
```

In Chapter 5 we introduced the formal concepts of deciding WS1S with non-deterministic automata and designed the algorithm for testing validity of a given formula φ. We then further extended this approach to an arbitrary k. In this section we will closely describe the practical implementation of this formal algorithm in the C++ language. The prototype of this procedure is going to be restricted to WS1S only.

For each level $1 \leq i \leq m$ we define two caches that will store the already computed results to avoid their repeated computation during the decision procedure. In case we get a cache miss, the value $\perp$ is returned instead. $BDDCache_i$ is a cache that maps states of level i to their corresponding MTBDDs representing their transition relations δ_i, i.e.

$$BDDCache_i : Q_i \to ((\Sigma_i \to 2^{Q_i}) \cup \{\perp\}). \tag{6.1}$$

Further we define the cache $StateCache_i$ for storing which states are final and which are non-final, i.e.

$$StateCache_i : Q_i \to \{Final, NonFinal, \perp\} \tag{6.2}$$

The core function for deciding WSkS is CheckForAcceptingState which checks whether there exists a reachable final state on level $0 \leq i \leq m$ from state q. This function is corresponding to the implementation of Algorithm 2 with use of function buildSuccessorTree for building successors of states and function `IsStateFinal` from Algorithm 3 for checking whether the state is final. The implementation is based on the workset algorithm and uses efficient apply operations over the MTBDDs of constructed successors.

Function CheckForAcceptingState(state q, level m)

Input: A state q of the automaton A_{φ_m}, a level of determinization m
Output: TRUE if there is a reachable accepting state from q, FALSE otherwise

```
1  workset := {q};
2  processed := workset;
3  while ∃q_m ∈ workset do
4  |   workset := workset \{q_m};
5  |   if IsStateFinal(q_m, m) then
6  |   |   return TRUE;
7  |   else
8  |   |   succ := buildSuccessorTree(q_m, m);
9  |   |   apply1 succ (λ x. if x ∉ processed ∧ ¬∃y ∈ processed : x < y
   |   |                        then workset := workset ∪{x});
10 |   |   processed := processed ∪ workset;
11 return FALSE;
```

The main Algorithm 3 for deciding WSkS is corresponding to the implementation of Algorithm 1. It first checks the cache whether a state has already been decided and else calls the function CheckForAcceptingState for checking if there exists a reachable final state.

We can further extend this procedure to deciding satisfiability of WSkS formulae by checking the validity of $\exists$-closed input formulae. This means that given the input we close the formula and its negation and test the validity of constructed formulae to decide their satisfiability. Further we can decide the formula according to the relationships between validity, satisfiability and unsatisfiability [8].

Function buildSuccessorTree(state q, level m)

Input: A state q of the automaton A_{φ_m}, a level of determinization m
Output: A successor of state q
Data: `BDDCache` mapping macro-states to their transition BDDs

```
1  succ := BDDCache[q];
2  if succ ≠ ⊥ then
3  |  return succ;
4  succ := ∅;
5  foreach r ∈ q do
6  |  childSucc := buildSuccessorTree(r, m − 1);
7  |  apply2 succ childSucc (λ x y. x ∪ y);
8  succ := project(succ, level);
9  BDDCache[q] := succ;
10 return succ;
```

The successor of a state can be constructed out of its children as described by the function buildSuccessorTree. This construction is also optimized with the use of the cache. After the MTBDDs for transitions of children of a state are built, we use the binary apply for doing the union of those MTBDDs to yield the MTBDD corresponding to the transition from the state. This MTBDD is further modified using the operation of projection.

6.3 Optimizations

During the implementation process we used the tool `valgrind` [3], especially its part `callgrind`, to profile the implementation in C++ and tried to identify weak spots of the application. One of the most frequent operations in the procedure are the relational operators on the classes representing macro-states used e.g. for pruning the state search, macro-state comparison and operations with worklist.

We propose to optimize this by using bitwise operations and bit array for storing the leaf states to ease some of the computation. There are several possibilities to use in C++ [16] and we have chosen the container with best performance, which is `boost::dynamic_bitset`. The results have shown that the performance of comparison between macro-states of first level of determinization got better by fair margin.

Besides some of the minor optimizations, we also extended the set of atomic formulae and tried to cache some of the intermediate results during the process to lessen the computation time. We will briefly describe these optimizations in the following subsections and the impact of the optimizations will be further discussed in following Chapter 7.

6.3.1 Extending set of atomic formulae

While the set of atomic formulae in the restricted syntax is indeed enough for defining the whole range of WSkS logic, automata corresponding to the flattened versions of some of atomic formulae (like $\leq$ for example) can be too large for processing and slow down the algorithm. As shown in Table 6.1 we can heavily reduce the size of the matrix automaton just by extending the set of the atomic formulae defined in 3.3 and construct special automata for frequently used operations and yielding smaller base automata in result.

Formula	quantifiers/alternations		automaton size [states]	
	before	after	before	after
$X \subseteq \mathbb{N}$	0	0	∞	$\max(X)+2$
$X = \{1, 4, 6\}$	0	0	24	8
$X = \{1, 3, 5, 7, 9\}$	0	0	93	11
$x \in X$	0	0	5	2
const $k \in X$	$k+1$	0	k+3	k+3
$0 \in X$	1	0	3	3
$1 \in X$	2	0	4	4
$x \leq y$	3/2	0	13	4

Table 6.1: Comparison of the basic set of atomic formulae with the extended set in size of the automaton, number of quantifiers and alternations. Note that ∞ denotes that number of states cannot be measured or described by mathematical equation.

Example 6.1. *Let us take the formula* $\varphi \stackrel{def}{=} x \leq y$ *for example. In the classical restricted syntax, this formula* φ *would be flattened as follows:*

$$x \leq y \stackrel{def}{\Leftrightarrow} \forall X.(y \in X \wedge (\forall z.z1 \in X \Rightarrow z \in X)) \Rightarrow x \in X \tag{6.3}$$

which corresponds to the automaton with 13 states. However, we can construct the following atomic automaton with 4 states only depicted in Figure 6.2, which means we can save around 70% *of states.*

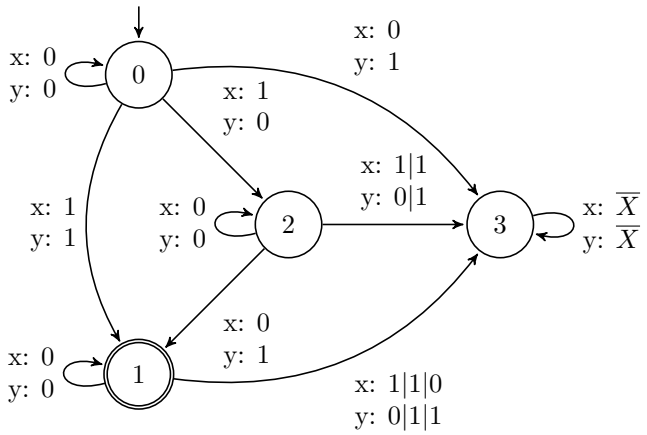


Figure 6.2: Atomic automaton corresponding to atomic formulae $x \leq y$

Even though some of the reductions can seem to be insignificant, due to their frequent occurrences in many practical formulae the overall reduction can be enormous in result. Another reduction comes from lessening the number of quantifications produced by creation of some temporary variables like for formula $\varphi \stackrel{def}{=} k \in X$ which after extending atomic formulae yields same number of states, however without quantifications, and hence decrease the number of projections of MTBDDs during the process.

Besides defining automata for more of the WSkS syntax connectives and atomic formulae, we could extend the syntax even further and create some of the more complex operations like the modulo or the predecessor predicate, just like MONA does, which are not specifically defined in WSkS logic. MONA [15] takes this even further and allows us to define external automata that can be included in the input specification.

Note that we can decide Presburger formulae [21] with the basic set of atomic formulae as shown for example in [10]. This approach yields big automaton in process with one alternation of negation for each addition in formula. However, we can construct a special automaton accepting the addition of two Presburger constants with 4 states only, thus greatly reducing the time needed to decide this family of formulae.

6.3.2 Cache

The caching of some of the intermediate products can be applied to several places in code. First we can cache the already computed results of deciding which of the macro-states are final or non-final. Also during the computation of successors of states, we can cache the MTBDDs that were already constructed before.

Generally, caching is a trade-off between time and space, which raises the questions how much will we store in the cache. One of the most frequent operations over MTBDDs is the union of posts of macro-states. By storing all of the intermediate results during the union of MTBDDs we can speed up the algorithm greatly on the expenses of using more memory.

With the design of algorithm, which is looking for existing example, i.e. searching for reachable state, in automaton corresponding to the formulae and its negation (to get counterexample) this means we do not have to compute and classify lots of states twice.

Chapter 7

Evaluation

In this chapter we are going to provide an evaluation of our tool `dWiNA`[1] — an implementation of the decision procedure described in the previous chapters. All the tests were performed in a virtualized environment with the Linux Ubuntu 12.10 operating system with 4 096 MB of operating memory and one virtual processor natively running on a laptop computer with 8 GB RAM memory and a dual core processor running on 2.5 GHz.

The performance of our prototype implementation was tested by measuring several different metrics and compared with the MONA tool which uses the deterministic tree automata instead of non-deterministic automata used in our approach and is heavily optimized (See Section 4.2) as well. The run tests were measuring the speed and size of the state space of the decision procedure depending on the size of the input formula and the number of quantifier alternations. The other tests were examining the impact of the used optimizations on the performance of our implementation, like the cache hit-miss ratio or the overall speed.

Note that we will use the symbol ∞ if either the test fails due to insufficient memory or it has been interrupted due a timeout, which was set to 5 minutes.

7.1 Parametric Horn formulae

Experiments done with our implemented tool were carried out on a specific parametric family of formulae in the following Horn form:

$$\varphi_n \stackrel{def}{=} \exists X \forall x_1 \dots x_n . \bigwedge_{i=1}^{n-1} (x_i \in X \Rightarrow x_{i+1} \in X) \tag{7.1}$$

The formulae in this family are closed and valid and suitable for our experiments since we can freely change the number of alternations in the formulae and the size of the resulting automaton depending on the quantifier prefix and the parameter n. Also it was further discovered that MONA is able to only decide formulae up to the value of $n = 15$ due to insufficient memory for storing MTBDDs. In results it is proposed by its authors that the logical approach designed in [9] proves to be better for solving this kind of formulae in contrary to the automata-based approach used by both MONA and `dWiNA`. However, we will show that our algorithm and the used library for manipulating non-deterministic automata [17] is far more memory efficient with handling MTBDDs and has no problems deciding these formulae and thus can fairly compete with the logical composition approach.

[1] deciding With Non-deterministic Automata

7.2 Comparison with MONA

We have run tests with the generated formulae of the form 7.1 for the parameter n ranging from 2 to 50 with various numbers of quantifier alternations bounded by the value of n ranging from 2 to 9 alternations per formula. The size of the base automaton (corresponding to the quantifier-free matrix) is heavily dependent on the number of alterations in the formulae as depicted in Table 7.1. Even number of alternations spawns one additional negation of the base automaton which then results in the disjunction of formulae that is handled as the union of automata. This way the automaton size is increasing heavily in the relation with parameter n.

n	**automaton [states]**	
	odd alternations	**even alternations**
3	18	30
4	27	144
5	36	648
6	45	3 240
7	54	15 336

Table 7.1: The base automaton size depending on the number of alternations in the formula and parameter n. The number of alternations can range from 1 to n, where even number spawns additional negation and results into the union of automata which heavily increases the resulting base automaton size.

In [9] it was proposed that this type of formulae is not suitable for automata-based approach and logical decomposition is a much better decision procedure. The results in Table 7.2 certainly proved that MONA has problems with deciding these formulae for $n \geq 15$, however, this is due to the enormous size of MTBDDs representing transitions in the automatona that are stored as an optimization and not a flaw in the automata-based approach, since our tool `dWiNA` has no problems with deciding formulae for a greater n.

n	**time [s]**		
	logic	`dWiNA`	`MONA`
10	0.01	0.01	0.12
12	0.01	0.01	0.89
13	0.01	0.01	2.28
14	0.01	0.01	5.53
15	0.01	0.02	15.06
16	0.01	0.02	∞
50	0.01	0.07	∞

Table 7.2: Time evaluation of deciding formulae of the form 7.1 in dependence on the parameter n for a fixed number of alternations (1).

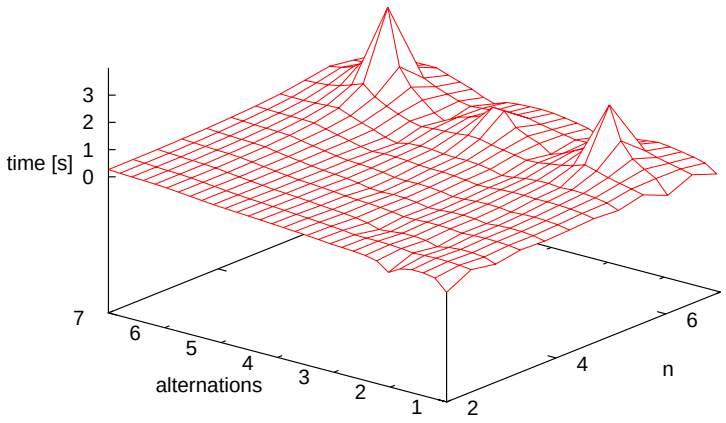


Figure 7.1: Speed performance of `dWiNA` based on the parameter n and the number of alternations in the formula. For even number of alternation the size of the base automaton rises heavily and thus increases the overall time needed to perform the decision procedure.

We further compared our tool with the MONA tool in dependence on the parameter n and the number of alternations. It is clear from Figure 7.1 that due to the huge differences in the base automaton size between odd and even alternations on the generated benchmark formulae the time needed for the computation is rather unsteady for our implementation. This is because of the inefficient representation of macro-states which makes some operations take too much time to compute, like pruning or state comparison. MONA, however, has no problem with alternations and yields steady times for all numbers of alternations showing that the full construction of the automaton along with minimizations is still efficient and gives good results. This is contrary to our base assumption that our *on-the-fly* approach should not have issues with alternations. Note that some combinations of alternations and parameter n cannot be done and were thus approximated in the shown mesh grid.

alternations	`dWiNA`	MONA
2	2.20	0.01
3	0.01	0.01
4	1.49	0.01
5	0.04	0.01
6	2.90	0.01

Table 7.3: Time comparison of `dWiNA` and MONA with fixed $n = 6$

In Figure 7.2 we show the alternate view on the data: the time comparison with MONA based on the size of the base automaton (our). Here we see that in dependence on the size of the automaton our approach is not that bad and can yield good results even though there are no automata reductions during the process. Even though MONA is far more consistent when it comes to the number of alternations of quantifiers in the formulae and has better computational results, in terms of the generated and evaluated search space our `dWiNA` is more efficient. Figure 7.3 shows that the *on-the-fly* approach generates much less states and needs only a portion of them for evaluation. This shows that our approach certainly has a potential to beat MONA even in time, but requires additional optimizations to take place.

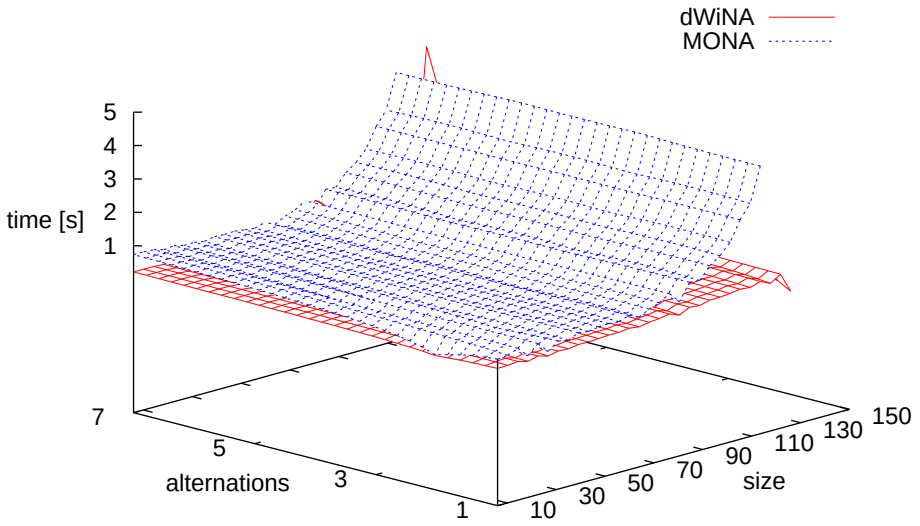


Figure 7.2: Alternative speed comparison of `dWiNA` and MONA based on the size of the automaton.

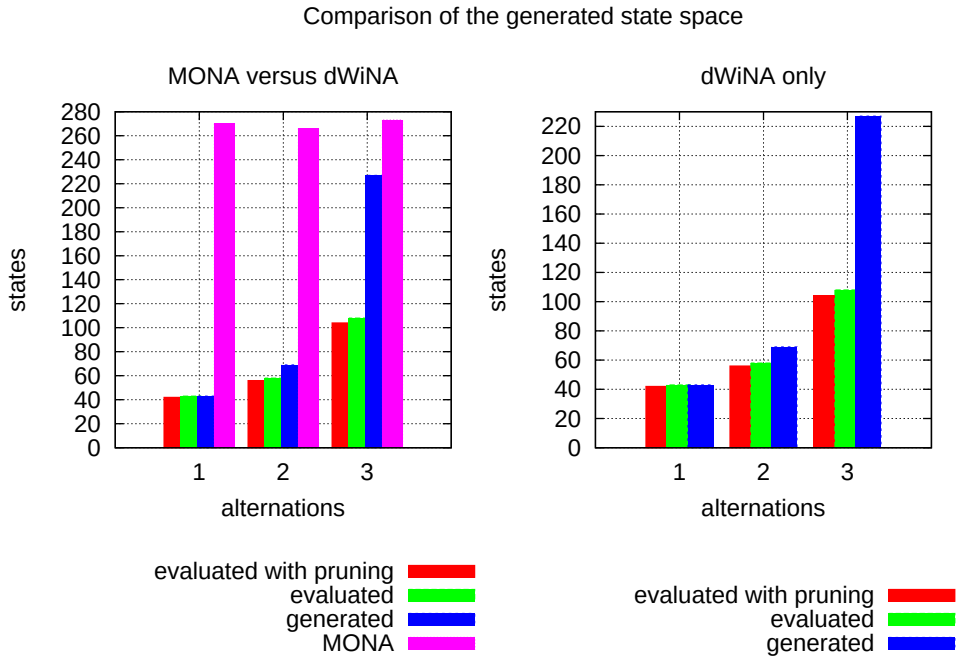


Figure 7.3: Comparison of the number of the generated/evaluated states between `dWiNA` and MONA on the base automaton with 36 states.

The other thing is that for many practical cases we do not need the fully expanded automaton to decide whether its language is universal (or empty) and usually any non-accepting (or accepting) suffices. Thus the *on-the-fly* approach is more suited for these kinds of problems.

7.3 The impact of the used optimizations

We will discuss the impact of optimizations developed on the performance of the implementation. Aside from some minor optimizations we introduced some of our implementation secrets in Section 6.3. The most notable ones is the pruning of the states during the search for the accepting states. The other ones is smarter flattening of the input formulae with an extended set of atomic formulae and the caching during the process.

We can see from Table 7.4 that all of these optimizations have a great impact on the performance. However, the greatest one is definitely the smart flattening since this way we are working with considerably smaller automata than with the basic atomic automata set. The pruning of the state space had a lower impact on this kind of formulae than expected as shown in Figure 7.3, but still is a great asset to the overall speed.

n	alternations	all [s]	no opt [s]	pruning [s]	smart flatten [s]	cache [s]
5	2	0.11	∞	24.05	0.49	25.20
	3	0.02	0.75	0.16	0.04	0.08
	4	0.14	∞	∞	1.09	13.71
	5	0.03	5.40	1.04	0.17	0.42
6	2	2.20	∞	∞	9.59	∞
	3	0.01	1.22	0.30	0.06	0.14
	4	1.49	∞	∞	11.15	∞
	5	0.04	17.7	2.34	0.28	2.47

Table 7.4: Speed comparison for various optimizations based on the number of alternations and the parameter n. The columns show what optimization was used with implementation and time it took to decide the given formula.

Other notable optimization is definitely the cache. Based on the result in Table 7.4 the impact of caching intermediate results during the decision process is high and for some n, where the implementation without optimizations fails, is even able to give a proper answer. While the cache that stores whether a given state is final or non-final is not overly used in the process, the cache for storing MTBDDs and especially some of the intermediate results during the construction of MTBDDs corresponding to the successors of states is used in more than 75% of cases. A more detailed look on the efficiency of the MTBDD cache is depicted in Table 7.5. The cache was tested on various formulae of the parameter n and the results are shown for macro-state levels up to 7.

Level	Cache-hit ratio
1	0.816
2	0.768
3	0.760
4	0.728
5	0.704
6	0.667

Table 7.5: Average cache-hit ratio for various level of determinizations for the MTBDD cache.

7.4 Discussion of results

In this chapter we have shown that the implementation can beat MONA in several aspects, namely the generated search space and for some cases the computation time to decide benchmark formulae. While MONA is indeed far more consistent with its results, we can see that for some kinds of formulae generating the whole state space can be excessive and to search for an accepting state in the automaton corresponding to formulae only a small fragment of the state space is required.

Our tool thus has a good potential of becoming a proper decision procedure for WSkS, but still needs some optimizations to yield more consistent results in practice.

Chapter 8

Conclusion

In this work we introduced the WSkS logic and some of its decision procedures and implementations. We have described the classical approach that uses deterministic automata, as well as the MONA tool that enhances this procedure by using several optimizations and discussed their complexity issues and problems they have to deal with.

Another approach was proposed that uses non-deterministic automata instead of deterministic ones. This makes use of recent developments in fields of non-deterministic automata algorithms, like universality checking or language inclusion, allowing us to implement a procedure similar to antichain-based testing [4] and search for rejecting or accepting states on-the-fly without need to construct the automaton corresponding to the given formula at all.

We implemented a prototype of the designed decision procedure that is able to handle a subset of WSkS formulae, namely formulae for $k = 1$, and studied the impact of the use of non-deterministic automata on several case studies. Out of the computational results on a set of formulae we identified some of the weak spots of the application and tried to optimize them to achieve better results.

We evaluated our tool `dWiNA` on a family of parametric formulae of the Horn form and compared it with MONA in several different aspects. We have shown that the *on-the-fly* approach generates only a portion of the state space in contrary to the classical deterministic approach. While MONA is more consistent with its results and has no problems with an excessive number of alternations, based on the size of the base automaton we were able to beat MONA even in the speed. The non-deterministic approach to deciding WSkS has indeed a great potential and may yield good results with future research.

We further propose some optimizations that could enhance the results. One of the weak spots of the implementation is the size of the base automaton corresponding to the quantifier-free matrix of the formula. MONA always performs minimization after every operation so it works with the smallest automata possible. We could reduce the size of non-deterministic automata through simulation computation followed by the downward reduction.

Most of the formulae used in practice share very similar subformulae that can be represented as a single state and reuse the constructed automaton by reindexing its variables. We propose to optimize the frontend of the procedure to use the Direct Acyclic Graph (DAG) instead of plain AST.

Bibliography

[1] MONA: Web pages of MONA. [online] Available on: <http://www.brics.dk/mona/>.

[2] Timbuk Reachability Analysis and Tree Automata Calculations. [online] Available on: <http://www.irisa.fr/celtique/genet/timbuk/>.

[3] Valgrind's Tool Suite. [online] Available on: <http://valgrind.org/info/tools.html>.

[4] Parosh A. Abdulla, Lukáš Holík, Yu-Fang Chen, Richard Mayr, and Tomáš Vojnar. When Simulation Meets Antichains (On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata). In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 6015, pages 158–174. Springer Verlag, 2010.

[5] Julius R. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly*, 6(1–6):66–92, 1960.

[6] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Denis Lugiez, Florent Jacquemard, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. Available on: <http://www.grappa.univ-lille3.fr/tata>, 2007.

[7] Laurent Doyen and Jean-Francois Raskin. Antichain Algorithms for Finite Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 2–22. Springer Berlin Heidelberg, 2010.

[8] Herbert B. Enderton. *A Mathematical Introduction to Logic*. 1979.

[9] Tobias Ganzow and Lukasz Kaiser. New Algorithm for Weak Monadic Second-Order Logic on Inductive Structures. In *Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 366–380. Springer Berlin Heidelberg, 2010.

[10] James Glenn and William Gasarch. Implementing WS1S via Finite Automata. In *Automata Implementation*, volume 1260 of *Lecture Notes in Computer Science*, pages 50–63. Springer Berlin Heidelberg, 1997.

[11] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. MONA: Monadic Second-Order Logic in Practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1995.

[12] Thomas A. Henzinger, Jean-Francois Raskin, and Martin De Wulf. Antichains: A New Algorithm for Checking Universality of Finite Automata. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer Berlin Heidelberg, 2006.

[13] Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. [online] Available on: <http://dinosaur.compilertools.net/yacc/>.

[14] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Available on: <http://www.brics.dk/mona/mona14.pdf>.

[15] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA Implementation Secrets. In *Workshop on Implementing Automata*, 2010.

[16] Derrick G. Kourie, Vreda Pieterse, Loek Cleophas, and Bruce W. Watson. Performance of C++ Bit-vector Implementations. In *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 242–250, 2010.

[17] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. The VATA Tree Automata Library. [online] Available on: <http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/>.

[18] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin Heidelberg, 2012.

[19] Alexander Meduna. *Automata and Languages: Theory and Applications*. Springer Verlag, 2005.

[20] Albert R. Meyer. Weak Monadic Second Order Theory of Succesor is Not Elementary-recursive. In *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154. Springer Berlin Heidelberg, 1975.

[21] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes Rendus du I congrés de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[22] Michael O. Rabin. Decidability of Second-Order Theories and Automata on Infinite Trees. *Transactions of the American Mathematical Society*, pages 1–35, 1969.

Appendix A

Contents of CD

In the main directory of the cd, there is a `cmake` source for compilation of the application.

Directory `/doc/thesis` Contains LaTeX sources with `Makefile` for the compilation of this thesis.

Directory `/examples` Contains examples written in MONA syntax for deciding. Ranging from simple examples used for testing functionality to more complex ones used for evaluation of the created code.

Directory `/include/vata` Contains headers needed to be included during the compilation of program for part of the functions that are used from the `libvata` library.

Directory `/src/app/DecisionProcedure` Contains main part of the application sources that does the conversion of given representation of formulae into the non-deterministic automaton and the procedure for deciding validity or satisfiability.

Directory `/src/app/Frontend` Contains part of the sources that does the parsing of the input formulae specification into inner representation in program

Directory `/src/libs` Contains external libraries that are used in application. Namely these are several libraries needed for MONA frontend and the `libvata` library for manipulating with NTA.

Appendix B

WSkS specification syntax

The following grammar describes supported subset of MONA syntax for the specification of verified formulae. It uses the classical BNF-like notation. For full syntax for MONA program consult the tool manual [14].

Specification

```
program ::= (header;)? (declaration;)+
header ::=  ws1s | ws2s
```

Declarations

```
declaration ::= formula
             |  var0 (varname)+
             |  var1 (varname)+
             |  var2 (varname)+
             |  'pred' varname (params)? = formula
             |  'macro' varname (params)? = formula
```

Formulae

```
formula ::= 'true' | 'false' | (formula)
         |  zero-order-var
         | ~formula
         | formula | formula
         | formula & formula
         | formula => formula
         | formula <=> formula
         | first-order-term = first-order-term
         | first-order-term ~= first-order-term
         | first-order-term < first-order-term
         | first-order-term > first-order-term
         | first-order-term <= first-order-term
         | first-order-term >= first-order-term
         | second-order-term = second-order-term
         | second-order-term = { (int)+ }
         | second-order-term ~= second-order-term
```

```
        | second-order-term 'sub' second-order-term
        | first-order-term 'in' second-order-term
        | ex1 (varname)+ : formula
        | all1 (varname)+ : formula
        | ex2 (varname)+ : formula
        | all2 (varname)+ : formula
```

First-order terms in WS1S

```
first-order-term ::= varname | (first-order-term)
                  |  int
                  | first-order-term + int
```

Second-order terms in WS1S

```
second-order-term ::= varname | (second-order-term)
                   |  second-order-term + int
```

Appendix C

Usage

The usage of the decision procedure tool is:

```
dWiNA [options] <filename>
```

`<filename>` is relative or absolute path to specification of WSkS formula as defined by syntax described in Appendix B. The options that can be further set are following:

- `-t` ,`--time` – prints elapsed time for decision procedure and further information about timing of procedure.
- `-d` ,`--dump-all` – dumps information about AST representation of given formula, symbol table, created automaton and etc.
- `-q` ,`--quiet` – suppress printing of information about decision process.

  `--reorder-bdd` – by default variables are reorder according to the prefix of the given formula. This can be suppressed by adding parameter `no` or random reordering can be done by option `random`.

C.1 Instalation

To compile the application run the following command from the main directory:

```
$ make release
```

To run the application use the following command or consult the usage:

```
$ ./dWiNA ./examples/formulae/in.mona
```

Appendix D

List of Atomic Formulae

We list in this Appendix automata corresponding to atomic formulae of WSkS logic used in decision procedure. These automata are further used for the construction of initial base automaton as described in Section 3. This appendinx is structured into two parts: first describes the basic set of atomic formulae defined for restricted syntax (see 3.3) and the other lists automata for the extension of syntax we are supporting to optimize size of used automata.

Note that all shown automata are non-deterministic and we use symbol $\overline{X}$ as a substitute for any symbol of Σ, i.e. we do not care about its value.

D.1 Atomic formulae for restricted syntax

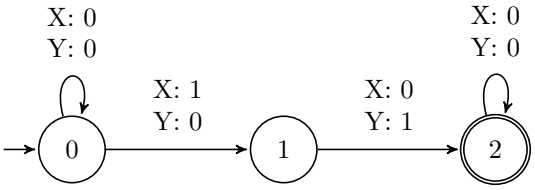


Figure D.1: Automaton corresponding to atomic formulae $X = Y1$, i.e. Y is successor of X.

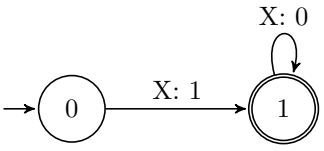


Figure D.2: Automaton corresponding to atomic formulae $X = \epsilon$.

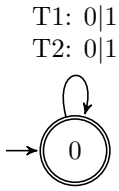


Figure D.3: Automaton corresponding to atomic formulae $T_1 = T_2$, where T_1 and T_2 are two second-order variables.

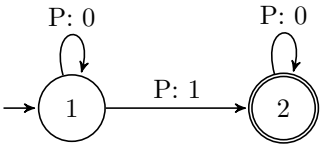


Figure D.4: Automaton corresponding to atomic formulae *Singleton*(P), i.e. that P is set containing exactly one element.

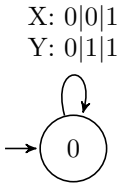


Figure D.5: Automaton corresponding to atomic formulae $X \subseteq Y$

D.2 Extending restricted syntax

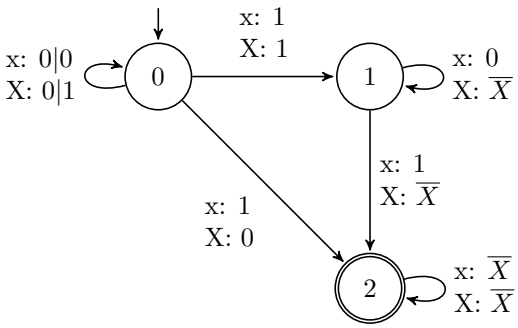


Figure D.6: Automaton corresponding to atomic formulae $x \in X$

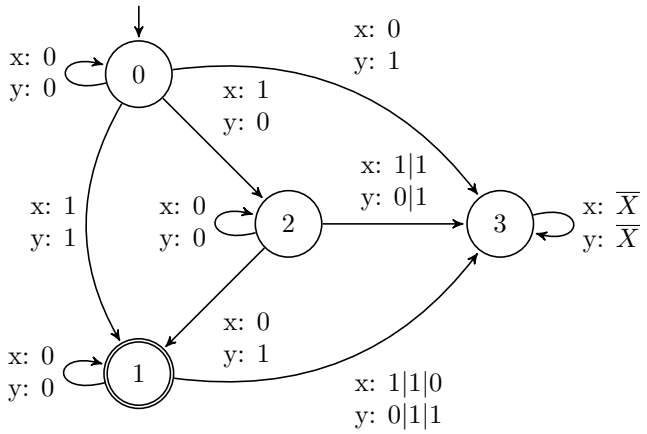


Figure D.7: Automaton corresponding to atomic formulae $x \leq y$

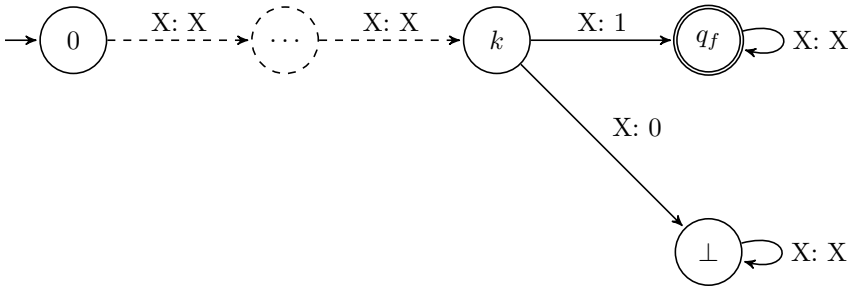


Figure D.8: Automaton corresponding to atomic formulae $\texttt{const}k \in X$

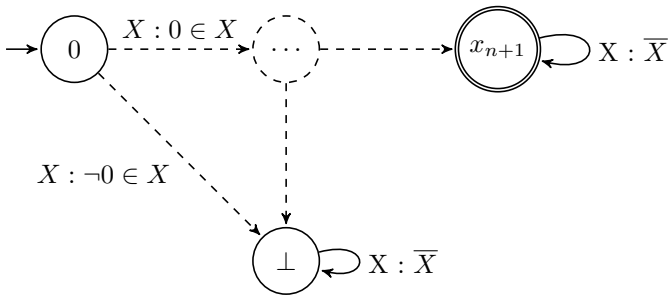


Figure D.9: Automaton corresponding to atomic formulae $X = \{x_1, \ldots, x_n\}$, for some ordering of integer constants $x_1 \leq \ldots \leq x_n$.