BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SIMULATION FOR SYMBOLIC AUTOMATA

SIMULACE PRO SYMBOLICKÉ AUTOMATY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **JURAJ SÍČ**
AUTOR PRÁCE

SUPERVISOR **Mgr. LUKÁŠ HOLÍK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Zadání bakalářské práce/19745/2016/xsicju00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Síč Juraj**
Obor: Informační technologie
Téma: **Simulace pro symbolické automaty**
Simulation for Symbolic Automata
Kategorie: Teoretická informatika

Pokyny:

1. Seznamte se s existujícími algoritmy pro výpočet simulace na konečných automatech a s problematikou symbolických automatů.
2. Navrhněte efektivní algoritmus pro výpočet simulace nad symbolickými automaty vycházející z algoritmů pro klasické automaty.
3. Analyzujte složitost nabo dokažte korektnost algoritmu, případně proveďte oboje.
4. Algoritmus implementujte a otestujte jeho efektivitu a škálovatelnost na vhodných datech (např. automaty ze související literatury, automaty vznikající při řešení verifikačních úloh a pod.). Porovnejte také míru minimalizace automatů dosaženou pomocí spočtené relace simulace s deterministickou minimalizací.

Literatura:

- Monika Rauch Henzinger, Thomas A. Henzinger, Peter W. Kopke:Computing Simulations on Finite and Infinite Graphs. Proc. of FOCS 1995, IEEE, pp. 453-462 (2013)
- Loris D'Antoni, Margus Veanes:Equivalence of Extended Symbolic Finite Transducers. Proc. of CAV 2013, Springer, LNCS 8044, pp. 624-639 (2013)
- Francesco Ranzato, Francesco Tapparo:An efficient simulation algorithm based on abstract interpretation. Inf. Comput. 208(1): pp. 1-22 (2010)

Pro udělení zápočtu za první semestr je požadováno:
- Body 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Holík Lukáš, Mgr., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2
L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstract

Symbolic automata are similar to classical automata with one big difference: transitions are labelled with predicates defined in separate logical theory. This allows usage of large alphabets while taking less space. In this work we are interested in computing simulation (a binary relation on states that language inclusion) for these automata. This can be then used for reducing the size of automata without the need to determinize them first. There exist few algorithms for computing simulation over Kripke structures, which were then altered to work over labeled transition systems and classical automata. We show how one of these algorithms can be modified for symbolic automata by using the partition of the alphabet domain that is compatible with the predicates labelling transitions and by using the possibilities of the alphabet theory.

Abstrakt

Symbolické automaty sú podobné klasickým automatom s jedným veľkým rozdielom: prechody sú značené predikátmi definovanými v oddelenej teórii. Toto umožňuje použiť veľké abecedy s použitím oveľa menšieho miesta. V tejto práci sa zaoberáme výpočtom simulácie (binárnej relácie nad množinou stavov, ktorá aproximuje jazykovú inklúziu) pre tieto automaty. Táto relácia sa dá potom použiť pri redukovaní počtu stavov bez nutnosti determinizácie. Existuje niekoľko algoritomv pre výpočet simulácie pre Kripkeho štruktúry, ktoré boli neskôr modifikované pre označené prechodové systémy a klasické automaty. V tejto práci ukážeme ako sa dá jeden z týchto algoritmov modifikovať pre symbolické automaty použitím rozkladu domény abecedy ktorý je kompatibilný s predikátmi značiacimi prechody a použitím možností teórie abecedy.

Keywords

symbolic automata, simulation, reduction of automata

Kľúčové slová

symbolický automat, simulácia, redukcia automatov

Reference

SÍČ, Juraj. *Simulation for symbolic Automata.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Holík Lukáš.

Simulation for symbolic Automata

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mgr. Lukáš Holík, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Juraj Síč
May 15, 2017

Contents

Chapter 1

Introduction

Finite automata (FAs) have many applications from text processing to formal verification. However, FAs have one big disadvantage: representing automata with big alphabets can take a lot of space. Another problem with big alphabets is that complexity of most of the algorithms for FAs depends on the number of symbols in the alphabet. This is also why using infinite alphabets is not possible for FAs. To resolve this problem, symbolic finite automata (SFAs) were introduced [24] (SFAs have a longer history, but in this work we use the formalization from [24]). Alphabets are separated from these automata and they are represented in their own theory. From this theory, we can choose a predicate that denotes a set of symbols from the alphabet and then use use it to label transitions in SFAs. This means that if there are transitions between the same states labelled with different symbols in FA, SFA can represent all these transitions with only one transition labelled with predicate that denotes these symbols. SFAs can then be used for tackling problems involving big alphabets such as processing regular expressions encoded in UTF-16 [26]. They are also used in combination with symbolic transducers[1] for sanitizer analysis[2] [27].

Applying algorithms for FAs to the symbolic setting directly is not always possible. Some classical automata constructions (product, difference and determinization) applied to SFAs are presented in [15]. In [8] authors tweaked two classical deterministic FAs minimization algorithms to SFAs and introduced another one that uses the symbolic representation to the full. While it was shown that finding minimal non-deterministic FA is a hard problem [18], a number of methods for reducing the number of states of non-deterministic FAs exists (without a guarantee of minimality). One of these methods is based on simulation [4, 5]. Simulation is a binary relation on states that tells us that if a state i simulates a state j, then for each transition going from j there is a corresponding transition going from i. Similar relation—bisimulation—can also be used for reducing the number of states. While both are similar, bisimulation is defined as an equivalence and simulation as preorder. This little difference can dramatically change the number of states of reduced automaton by using either of these relations and finding simulation is then an important problem (this difference also makes this problem harder). The classical algorithm for computing bisimulation on FAs is Paige-Tarjan's relational coarsest partition algorithm [20]. Algorithms for computing bisimulation on SFAs were investigated in [9]. State reduction of automata is not the only application of simulation. It can also be used in solving the language inclusion problem—when we need to check whether the languages accepted by automata are

[1] automata that also have output
[2] string transformation routines

in inclusion. Simulation underapproximates language inclusion, therefore it can be used to detect that the inclusion holds [1].

In this text we are concerned with finding an effective algorithm for computing simulation for SFAs. There are several algorithms for computing simulation for Kripke structures[3] where the first algorithm with good time complexity is the algorithm by Henziger, Henziger and Kopke [14]. Other, more effective, simulation algorithms are derived from this and they usually compute simulation preorder by working with equivalence classes of the current approximation of simulation instead of with individual states. A summary of these can be found in [21]. Independently of this, Ilie, Navarro and Yu [17] developed simulation algorithm for FAs which is very similar to the algorithm in [14]. Because they used simpler data structures, we use this algorithm as a basis for the simulation algorithms for SFAs.

The structure of this thesis is as follows. In Chapter 2 we give preliminaries to (symbolic) automata theory and simulation. In Chapter 3 we explain the workings of algorithm introduced in [17] and then we introduce three new algorithms for computing simulation for SFAs based on this algorithm. In Chapter 4 we thoroughly evaluate these algorithms on SFAs created from regular expressions over UTF-16 alphabet and during the decision procedure of weak-monadic second order logic of one successor and compare them with deterministic minimization of SFAs.

[3] a transition system where states are labelled, not transitions

Chapter 2

Automata Theory

In this chapter, we give the necessary theory that will be used in the next chapters. Firstly, we give definitions of classical finite automata and then we introduce effective Boolean algebras with symbolic finite automata and their properties. We also give overview of the concept called mintermization for effective Boolean algebras. At the end of the chapter, we explain what simulations on these automata are and how they can be used for reducing the number of states.

2.1 Classical Automata

Definition 2.1. *Finite automaton* (FA) is a tuple $N = (Q, \Sigma, \Delta, I, F)$ where Q is a finite set of *states*, Σ is an *alphabet*, $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition* relation, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states.

A transition $(q, a, p) \in \Delta$ where $q, p \in Q, a \in \Sigma$ is also denoted by $q \xrightarrow{a}_N p$ or $q \xrightarrow{a} p$ if N is clear from the context. For $a \in \Sigma$, we define a transition function $\overrightarrow{\Delta}_a \colon Q \to 2^Q$ where $p \in \overrightarrow{\Delta}(q, a)$ iff $q \xrightarrow{a} p$. Furthermore, we define a reverse transition function $\overleftarrow{\Delta}_a(p) = \{\, q \mid p \in \overrightarrow{\Delta}_a(q) \,\}$.

Elements of alphabet Σ are called characters and finite sequence of characters are called words, where ϵ denotes the empty word. The set of all words is denoted by Σ^*. The empty word ϵ is accepted from every final state $f \in F$. A word $w = a_1 a_2 \dots a_n$ is accepted from a state $q \in Q$ if there exists a sequence of states $r_1, r_2, \dots, r_{n+1}$ such that $r_1 = q$, $r_{n+1} \in F$ and $r_i \xrightarrow{a_i} r_{i+1}$ for every $i = 1, \dots, n$. The set of all words accepted from state q is denoted by $\mathcal{L}_M(q)$. The language accepted from a set of states P is $\mathcal{L}_M(P) \equiv \bigcup_{q \in P} \mathcal{L}_M(q)$ and the language accepted by the automaton M is $\mathcal{L}(M) \equiv \mathcal{L}_M(I)$. We now define an important property of completeness of FAs:

Definition 2.2. We say that FA $N = (Q, \Sigma, \Delta, I, F)$ is *complete* if for all states $q \in Q$ and characters $a \in \Sigma$, there exists a state p where $p \xrightarrow{a} q$.

2.2 Symbolic Finite Automata

Before defining symbolic automata, we firstly need to formally define the notion of an effective Boolean algebra. We use definitions from [24].

Definition 2.3. An effective Boolean algebra is a tuple $\mathcal{A} = (\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where:

- $\mathfrak{D}$ is a non-empty set called the *domain*,
- Ψ is a set of *predicates* closed under $\vee, \wedge, \neg$ where $\bot, \top \in \Psi$ and
- $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ is the *denotation function* where $[\![\bot]\!] = \emptyset, [\![\top]\!] = \mathfrak{D}$ and for all $\varphi, \psi \in \Psi, [\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!], [\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$ and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$.

For $\varphi \in \Psi$, we write $\mathit{IsSat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$ and say that φ is *satisfiable*. We now give some examples of effective Boolean algebras.

Example 2.4. Let S be some set. A trivial example of an effective Boolean algebra is the powerset algebra $\mathcal{A}_S$ where $\mathfrak{D}_{\mathcal{A}_S} = S$, $\Psi = 2^S$, $\bot = \emptyset$, $\top = S$, $\vee, \wedge$ and $\neg$ are the set operations union, intersection and complement, respectively, and $[\![\_]\!]$ is the identity function ($[\![A]\!] = A$ for $A \subseteq S$). ◊

Example 2.5. For $k > 0$, the *BDD algebra* $\mathbf{BDD}_k$ is an effective Boolean algebra whose domain is the set of non-negative integers smaller than 2^k (their binary representation has k bits) and predicates are binary decision diagrams (BDDs) [3] of depth k. BDDs are "binary tree structures" with two terminal nodes $\top$ and $\bot$. Every non-terminal node has two child nodes with edge labelled by either 0 or 1 (see Figure 2.1 for an example of a BDD). These structures are usually used as a succinct representation of Boolean functions, which allow for an efficient polynomial time implementation of logical operations . Boolean operations of the algebra $\mathbf{BDD}_k$ correspond directly to these BDD operations. The denotation of a BDD is the set of all numbers whose binary representation corresponds to a solution of the BDD. For example, the denotation of the BDD β from Figure 2.1 is $[\![\beta]\!] = \{3, 4, 5, 6, 7\}$, because all numbers whose binary representation is $1**$ or 011 are solutions of β. ◊

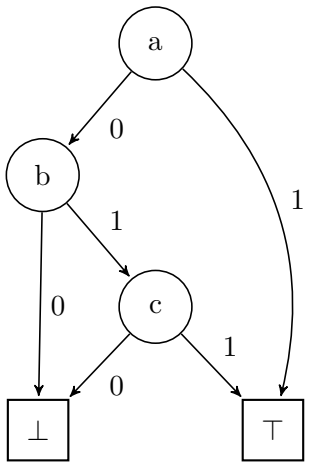


Figure 2.1: A binary decision diagram β.

Example 2.6. In this example we illustrate an effective Boolean algebra over first-order logic. Let $\mathbf{FOL}_{\mathfrak{D},\mathcal{T}} = (\mathfrak{D}, \Psi, [\![\_]\!]^{\mathcal{T}}, \bot, \top, \vee, \wedge, \neg)$ be the *first-order logic algebra* where $\mathcal{T}$ is some first-order theory. Informally, the first-order theory is the set of first-order formulas (called axioms) with constants, functions and predicates. Formula evaluates to true wrt. to some theory if also all axioms evaluate to true. See [2] for more information. Predicates in Ψ are then formulas containing constants, functions and predicates from $\mathcal{T}$ with one free variable x. A variable is free if it is not bounded by quantifier (x is free in the formula

$x < 5$ but not in the formula $\forall x.x < 5$). The operations of $\mathbf{FOL}_{\mathfrak{D},\mathcal{T}}$ $(\vee, \wedge, \neg)$ are classical logical connectives and $\top$, $\bot$ are formulas $x = x$ and $x \neq x$ respectively.

Interpretation of a first-order logic formula φ is a tuple (U, α) where U is a non-empty set and the assignment α maps constant, function and predicate symbols in φ to elements, functions and predicates over U. It also maps variables to elements in U. The denotation function $[\![\varphi(x)]\!]_{\mathcal{T}}$ is then the set of all elements $d \in \mathfrak{D}$ for which exists interpretation $I = (\mathfrak{D}, \alpha)$ where x is mapped to d in α and $\varphi(x)$ evaluates to true under I wrt. $\mathcal{T}$.

In this work we only use $\mathbf{FOL}_{\mathbb{Z},\mathcal{T}_\mathbb{Z}}$ where $\mathcal{T}_\mathbb{Z}$ is the theory of integers (as defined in [2]). What is important for us is that $<$ and divisibility by constant are both predicates in this theory and so formulae like $x < 5$ and $isEven(x)$ can be both predicates in effective Boolean algebra (where $[\![x < 5]\!]^{\mathcal{T}} = \{\ldots, -1, 0, 1, 2, 3, 4\}$, $isEven(x)$ denotes all even integers and, similarly, we have the predicate $isOdd(x)$ denoting all odd numbers). ◊

The algebra from Examples 2.5 can be used in practice as an API with corresponding methods implementing the operations $(\vee, \wedge, \neg$ and $IsSat\,(\_))$. The algebra from Example 2.6 can also be used for some theories with Satisfiability Modulo Theories (SMT) solver which can decide satisfiability of a quantifier free formula in some first-order theory.

We are now prepared to define symbolic finite automata. Informally, these automata have transitions labelled by predicates which denote subsets of elements from $\mathfrak{D}_\mathcal{A}$. $\mathfrak{D}_\mathcal{A}$ has then the same role as the alphabet for FA, but because one predicate can denote potentially infinite set of domain elements, symbolic finite automata can be used with a big or infinite alphabet while keeping a compact form.

Definition 2.7. *Symbolic finite automaton* (SFA) is a tuple $M = (Q, \mathcal{A}, \Delta, I, F)$ where Q is a finite set of *states*, $\mathcal{A}$ is an *effective Boolean algebra*, $\Delta \subseteq Q \times \Psi_\mathcal{A} \times Q$ is a *transition* relation, $I \subseteq Q$ is a set of initial states and $F \subseteq Q$ is a set of final states.

Example 2.8. In Figure 2.2 you can see an example of an SFA $M = (\{q_0, q_1, q_2, q_3\}, \mathbf{FOL}_{\mathbb{Z},\mathcal{T}_\mathbb{Z}}, \{(q_0, x < 5, q_1), (q_0, x > 5, q_1), (q_1, \top, q_1), (q_1, isOdd(x), q_2), (q_2, isEven(x), q_1), (q_2, isOdd(x), q_2), (q_3, x = 4, q_2)\}, \{q_0\}, \{q_2\})$ accepting the language $\mathcal{L}(M) = \{l \mid l$ is a list of at least two integers which does not start with 4 and ends with an odd number$\}$. The automaton M uses the effective Boolean algebra $\mathbf{FOL}_{\mathbb{Z},\mathcal{T}_\mathbb{Z}}$ from Example 2.6. ◊

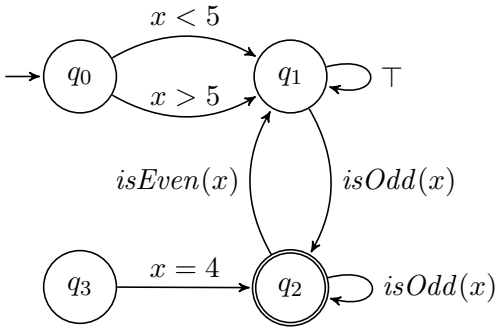


Figure 2.2: Symbolic finite automaton M.

Similar to FAs, a transition $(q, \psi, p) \in \Delta$ is denoted by $q \xrightarrow{\psi}_M p$ or $q \xrightarrow{\psi} p$ if M is clear from the context. Because the set of domain elements is in the role of the alphabet, we use the same terminology: elements of $\mathfrak{D}_\mathcal{A}$ are called characters, finite sequence of characters are called words, $\mathfrak{D}_\mathcal{A}^*$ denotes the set of all words and empty word is denoted by ϵ. We can

then say that for $a \in \mathfrak{D}_{\mathcal{A}}$, $q, p \in Q$, $q \xrightarrow{a}_M p$ is a transition $q \xrightarrow{\psi}_M p$ if $a \in [\![\psi]\!]$. This allows us to define accepted words from a state and by an automaton in the same way as we have defined them for FAs.

Next we define several important properties of SFAs which we later use and for each of them we provide an algorithm for transforming an SFA to a one satisfying the property.

Definition 2.9. We say that M is *complete* if for all states $q \in Q$ and characters $a \in \mathfrak{D}_{\mathcal{A}}$, there exists a state p where $p \xrightarrow{a} q$.

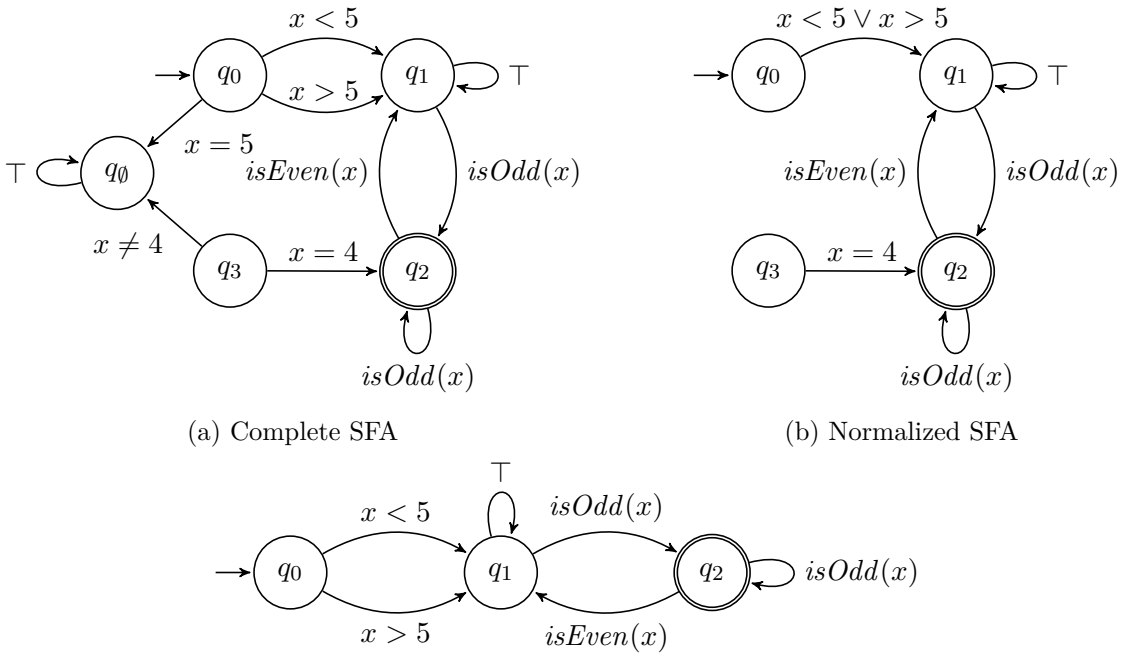
To make any SFA complete, we first need to add a new sink state $q_\emptyset$ to Q. After that, for each state $q \in Q$, we add a transition $(q, \varphi, q_\emptyset)$, where $\varphi = \neg \bigvee_{\exists p.q \xrightarrow{\psi} p} \psi$, to Δ if $IsSat(\varphi)$. The new automaton accepts the same language as the original automaton M.

Definition 2.10. We say that M is *normalized* if for all $q, p \in Q$ there exists at most one transition from q to p.

Normalization is straightforward. If there exist two states p, q with two distinct transitions $q \xrightarrow{\psi} p, q \xrightarrow{\varphi} p$, we replace these transitions with one transition $q \xrightarrow{\psi \vee \varphi} p$. Again, the language accepted by the new automaton is clearly the same.

Definition 2.11. We say that M is *clean* if for all states $q \in Q$, q is reachable from some $p \in I$, and for every $p \xrightarrow{\varphi} q$, φ is satisfiable. State q is reachable from p if $q = p$ or there exists a sequence of transitions $r_{i-1} \xrightarrow{a_i} r_i$ for $i \in \{1, 2, \ldots, n\}, n \in \mathbb{N}$ where $r_0 = p$ and $r_n = q$.

Cleaning of SFA is also not complicated. First, we remove all transitions $p \xrightarrow{\varphi} q$ where φ is not satisfiable and after that, we remove unreachable states.



(a) Complete SFA

(b) Normalized SFA

(c) Clean SFA

Figure 2.3: SFAs with different properties accepting the same language.

Example 2.12. Figure 2.3 illustrates the properties introduced above on the SFA M from Example 2.2. ◊

Definition 2.13. We say that M is *global minterm normalized* if for every two transitions $q_1 \xrightarrow{\varphi_1} p_1, q_2 \xrightarrow{\varphi_2} p_2$, $[\![\varphi_1]\!] = [\![\varphi_2]\!]$ or $[\![\varphi_1]\!] \cap [\![\varphi_2]\!] = \emptyset$.

Definition 2.14. We say that M is *local minterm normalized* if for every state q and every two transitions $q \xrightarrow{\varphi_1} p_1, q \xrightarrow{\varphi_2} p_2$, $[\![\varphi_1]\!] = [\![\varphi_2]\!]$ or $[\![\varphi_1]\!] \cap [\![\varphi_2]\!] = \emptyset$.

In the next section we explain what the minterms are, how to compute them and how to use them to transform any SFA to SFA with either of these properties.

2.2.1 Mintermization

In this section we introduce the concept of minterms [15], which we then use for transforming SFAs to their minterm normalized forms. We also give an overview of the algorithm for computing minterms introduced in [8]. Then we give an algorithm for transforming an SFA to its global minterm normalized form (this form was used indirectly in [8] for computing minimal deterministic SFA) and local minterm normalized form (our newly introduced form).

Definition 2.15. Let $\mathcal{A} = (\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ be an effective boolean algebra and $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_n\} \subseteq \Psi$ a non-empty finite set of predicates. Predicate $\bigwedge_{i=1}^{n} \psi_i$ where ψ_i is either φ_i or $\neg\varphi_i$ is called *minterm* generated from Φ. The set of all satisfiable minterms generated from Φ is denoted by $\mathit{Minterms}\,(\Phi)$, that is

$$\mathit{Minterms}\,(\Phi) = \left\{ \psi = \bigwedge_{1 \leq i \leq n} \psi_i \;\middle|\; \forall i \in \{\,1, \ldots, n\,\}\,.\,(\psi_i \in \{\,\varphi_i, \neg\varphi_i\,\}) \wedge \mathit{IsSat}\,(\psi) \right\}.$$

We also say that the minterm ψ is created from the predicate $\varphi \in \Psi$ if in the conjunction defining the minterm ψ, φ was used in its non-negated form.

Example 2.16. This example illustrates the notion of minterms. We use the algebra $\mathbf{FOL}_{\mathbb{Z},\mathcal{T}_{\mathbb{Z}}}$ and $\Phi = \{\,x < 5, x > 1, x > 8\,\}$. Let us list all minterms generated from Φ:

$$\psi_1 = x < 5 \wedge x > 1 \wedge x > 8$$

$$\psi_2 = x < 5 \wedge x > 1 \wedge \neg(x > 8)$$

$$\psi_3 = x < 5 \wedge \neg(x > 1) \wedge x > 8$$

$$\psi_4 = x < 5 \wedge \neg(x > 1) \wedge \neg(x > 8)$$

$$\psi_5 = \neg(x < 5) \wedge x > 1 \wedge x > 8$$

$$\psi_6 = \neg(x < 5) \wedge x > 1 \wedge \neg(x > 8)$$

$$\psi_7 = \neg(x < 5) \wedge \neg(x > 1) \wedge x > 8$$

$$\psi_8 = \neg(x < 5) \wedge \neg(x > 1) \wedge \neg(x > 8)$$

Because minterms ψ_1, ψ_3, ψ_7 and ψ_8 are not satisfiable, $\mathit{Minterms}\,(\Phi) = \{\,\psi_2, \psi_4, \psi_5, \psi_6\,\}$. The minterm ψ_2 is created from predicates $x < 5$ and $x > 1$. ◊

Note that the number of minterms is at worst exponential to the number of predicates in Φ, that is, $|\mathit{Minterms}\,(\Phi)| \leq 2^{|\Phi|}$. It can be easily shown (by induction over the number of predicates in Φ) that the set $P = \{\, [\![\psi]\!] \mid \psi \in \mathit{Minterms}\,(\Phi) \,\}$ is a partition of $\mathfrak{D}_{\mathcal{A}}$. Because of this, for each $a \in \mathfrak{D}_{\mathcal{A}}$ there exists *exactly* one $\psi \in \mathit{Minterms}\,(\Phi)$ where $a \in [\![\psi]\!]$. This also means that if Π is the set of all minterms created from φ, $[\![\varphi]\!] = [\![\bigvee_{\psi \in \Pi} \psi]\!]$.

To compute the set $\mathit{Minterms}\,(\Phi)$ for some set of predicates Φ we use the algorithm introduced in [8]. This algorithm uses a structure named predicate tree, which is a binary tree where values of nodes are predicates. Let us define—for a node n with the value φ—a recursive function $\mathit{Refine}(\psi)$:

1. If n is not a leaf, call $\mathit{Refine}(\psi)$ for the left and the right child of n.
2. If n is a leaf, create a left child with the value $\varphi \wedge \psi$ and a right child with the value $\varphi \wedge \neg\psi$.

For every leaf m with a value φ_m of the tree where n is the root, this function splits m to two new leaves with values $\varphi_m \wedge \psi$ and $\varphi_m \wedge \neg\psi$. We can then easily get $\mathit{Minterms}\,(\Phi)$ by starting with a predicate tree with only the root node with the value $\top$ and for each $\psi \in \Phi$ calling the function $\mathit{Refine}(\psi)$ for the root node. Then the values of the leaves are the minterms and by checking their satisfiability we get the set $\mathit{Minterms}\,(\Phi)$.

Example 2.17. In Figure 2.4 we give an example of predicate tree generated from the set $\Pi = \{\, x < 5, x > 1 \,\}$. We start with the tree with only the root node. We then call $\mathit{Refine}(x < 5)$ for the root value and get the second level of the tree. After that, we again call $\mathit{Refine}(x > 1)$ for the root and we get the final form of the predicate tree. ◊

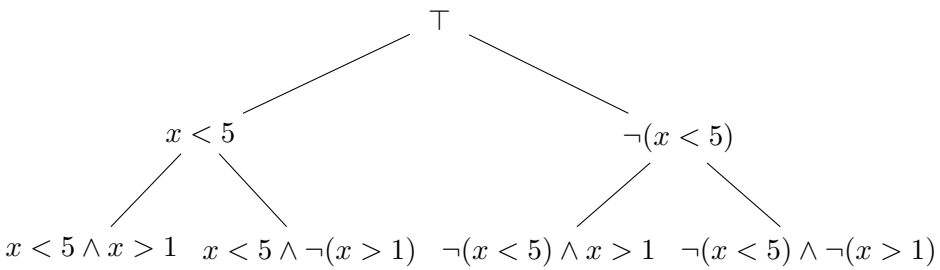


Figure 2.4: Predicate tree created by using *Refine* function.

Because we do not need all $2^{|\Phi|}$ minterms, but only those that are satisfiable, we can check during the computation whether there is any point in creating both children (or either). This is why we define for a node n with the value φ a function $\mathit{BetterRefine}(\psi)$:

1. If n does not have any children, then
   (a) if both $\varphi \wedge \psi$ and $\varphi \wedge \neg\psi$ are satisfiable, create both children where the left child has the value $\varphi \wedge \psi$ and the right child has the value $\varphi \wedge \neg\psi$,
   (b) if only $\varphi \wedge \psi$ is satisfiable (this means that $[\![\varphi]\!] \subseteq [\![\psi]\!]$, therefore $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!]$), create the left child with the value φ,
   (c) if only $\varphi \wedge \neg\psi$ is satisfiable (this means that $[\![\varphi]\!] \subseteq [\![\neg\psi]\!]$, therefore $[\![\varphi \wedge \neg\psi]\!] = [\![\varphi]\!]$), create the right child with the value φ.
2. If n has only one child, call $\mathit{BetterRefine}(\psi)$ for it.

3. If n has both children, then

   (a) if both $\varphi \wedge \psi$ and $\varphi \wedge \neg\psi$ are satisfiable, call *BetterRefine*(ψ) for both children,

   (b) if only $\varphi \wedge \psi$ is satisfiable, create for every leaf of the subtree with n as the root a left child with the same value as in the leaf,

   (c) if only $\varphi \wedge \neg\psi$ is satisfiable, do the same thing as in 3b, but the new nodes are added as the right children of the leaves.

The first step when the node does not have any children (the node is a leaf) is similar to the step 1 of *Refine* function, but we create only the children that are satisfiable. As a further optimization, in order to simplify the predicates, we replace $\varphi \wedge \psi$ by φ if only $\varphi \wedge \psi$ is satisfiable (and $\varphi \wedge \neg\psi$ is not), because this means that for every $a \in [\![\varphi]\!]$, a is also an element of $[\![\psi]\!]$. Similarly, we replace $\varphi \wedge \neg\psi$ by φ. This means that, strictly speaking, we now do not compute minterms, but predicates equivalent to them (that is predicates that denote the same set of domain elements).

In the third step when both children are present we either refine the children (3a) or, if only one of the predicates $\varphi \wedge \psi$ and $\varphi \wedge \neg\psi$ is satisfiable, we "extend" the subtree whose root node is the node that is being refined. This extension only creates a new level in the subtree, but values of the leaves do not change. The reason for this is that we can now easily find all predicates from Φ that each minterm was created from by following the path from root to the minterm. This is better explained in Example 2.18. Also, because in this step we create nodes with one child, we need to add step 2 where we refine a node with only one child. Because the child has the same value as the refined node, we just refine it.

Again, we get *Minterms* (Φ) by starting with predicate tree with only the root node with the value $\top$ and then refining the root node by each predicate in Φ. As we mentioned, we get in the leaves the predicates that are equivalent to minterms.

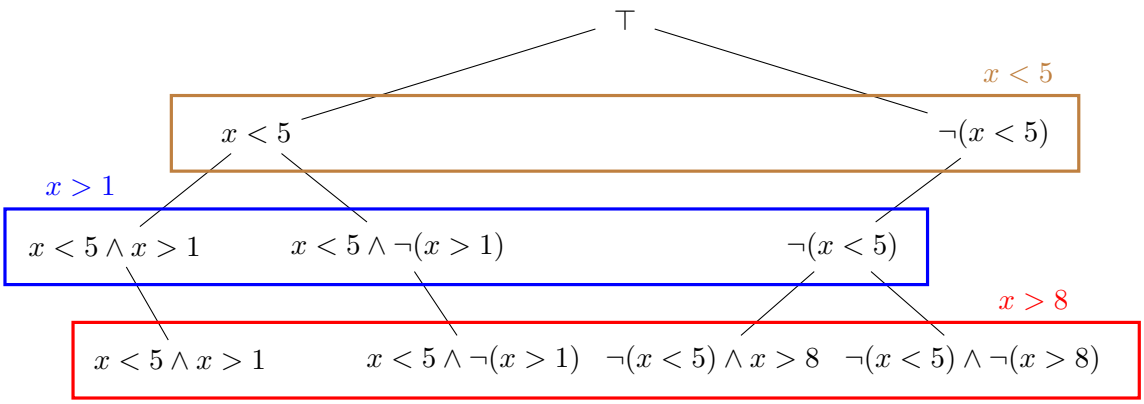


Figure 2.5: Predicate tree created by using *BetterRefine* function.

Example 2.18. In this example, we show how the refining of predicate tree works, using the set Φ from Example 2.16. We start with a predicate tree with only one node with the value $\top$. Then by refining this tree with $x < 5$, we get the second level of the tree shown in Figure 2.5 (the brown one). After refining with $x > 1$, the third level is added. Notice that because $\neg(x < 5) \wedge \neg(x > 1)$ is unsatisfiable, the value $\neg(x < 5)$ is copied to the created child in the left subtree. After refining with the last predicate $x > 8$, we get the final form of the predicate tree. In the right subtree, after checking that only $x < 5 \wedge \neg(x > 8)$ is

satisfiable, the previous leaves in level tree are "extended" to the right. Every predicate that refines the predicate tree then adds a new level to this tree. The values of the leaves are equivalent to the minterms ψ_2, ψ_4, ψ_5, and ψ_6. Also notice that each minterm ψ becomes the left leaf if ψ is created from the predicate φ that is refining the tree or the right leaf if ψ is not created from φ. This and the fact that each refined predicate adds a new level to the tree means that after we create the predicate tree we can look at the path from the root to the minterm to find the predicates from which the minterm is created. Let us explain on an example. The path from the root to the minterm $x < 5 \wedge \neg(x > 1)$ first goes left, so this minterm is created from $x < 5$. Then it turns right and we can see that this minterm is not created from $x > 1$. At last, the path goes left again and so we can say that the minterm is created only from the predicate $x < 5$. $\Diamond$

As promised we now explain how minterms can be used to transform any SFA $M = (Q, \mathcal{A}, \Delta_M, I, F)$ to its *global minterm normalized* or *local minterm normalized* form. Let

$$\mathit{Minterms}\,(M) \equiv \mathit{Minterms}\left(\{\,\psi \mid \exists q \xrightarrow{\psi}_M p\,\}\right)$$

be a set of minterms generated by the set of all labels occurring in the automaton M. Then SFA $M_G = (Q, \mathcal{A}, \Delta_{M_G}, I, F)$ where

$$\Delta_{M_G} = \{\,(q, \psi, p) \mid \psi \in \mathit{Minterms}\,(M) \wedge \exists q \xrightarrow{\varphi}_M p.(\mathit{IsSat}\,(\varphi \wedge \psi))\,\} \tag{2.1}$$

is the global minterm normalized form of M. That is, we have replaced every transition $q \xrightarrow{\varphi}_M p$ with transitions $q \xrightarrow{\psi_1}_{M_G} p, q \xrightarrow{\psi_2}_{M_G} p, \ldots q \xrightarrow{\psi_n}_{M_G} p$ where $\psi_1, \psi_2, \ldots, \psi_n$ are all minterms created from φ. This is why the possibility of determining the predicates that the minterm was created from in the predicate tree is useful. Because $[\![\varphi]\!] = [\![\bigvee_{1 \le i \le n} \psi_i]\!]$, $\mathcal{L}(M) = \mathcal{L}(M_G)$.

Local mintermization is similar; let for $q \in Q$,

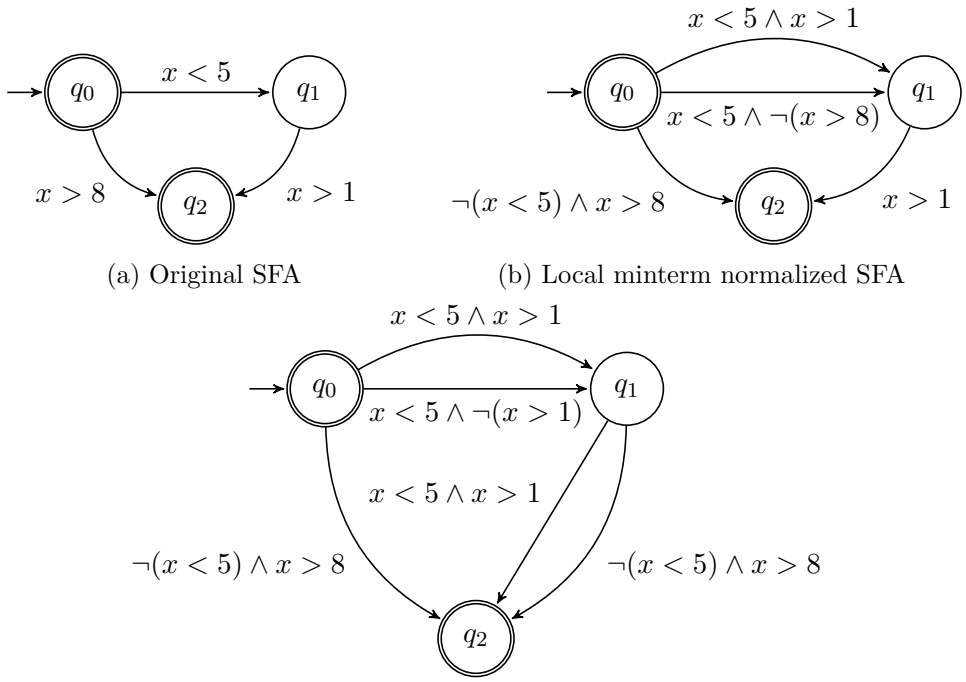
$$\mathit{Minterms}\,(q) \equiv \mathit{Minterms}\left(\{\,\psi \mid \exists q \xrightarrow{\psi}_M p\,\}\right),$$

that is, we get minterms generated by the set of labels occurring in transitions with the source state q. Then the SFA $M_L = (Q, \mathcal{A}, \Delta_{M_L}, I, F)$ where

$$\Delta_{M_L} = \{\,(q, \psi, p) \mid \psi \in \mathit{Minterms}\,(q) \wedge \exists q \xrightarrow{\varphi}_M p.\,(\mathit{IsSat}\,(\varphi \wedge \psi))\,\} \tag{2.2}$$

is the local minterm normalized form of M. Again, we have split transitions into minterm-labelled transitions, but only locally for each state. This usually results in a much smaller blow-up in the number of transitions than in global minterm normalized form.

Example 2.19. In this example we compare global and local minterm normalized forms created from the automaton $M = (\{q_0, q_1, q_2\}, \mathbf{FOL}_{\mathbb{Z},\mathcal{T}_\mathbb{Z}}, \{(q_0, x < 5, q_1), (q_0, x > 8, q_2), (q_1, x > 1, q_2)\}, \{q_1\}, \{q_3\})$. Labels are chosen from the set Φ from Example 2.16. In Figure 2.6 you can see the automaton M and local and global minterm normalized automata created from M. This example also shows that while the local automaton has 4 transitions, the global one has 5. If there were more transitions, this number would only grow. $\Diamond$



(a) Original SFA

(b) Local minterm normalized SFA

(c) Global minterm normalized SFA

Figure 2.6: Local and global minterm normalization.

2.3 Simulation

Having defined FAs and SFAs we can now introduce a simulation on automata. Firstly, we give the definition of the simulation for FAs and SFAs and then we show why this relation is important and where it can be used. In the next paragraphs let $N = (Q, \Sigma, \Delta_N, I, F)$ be an FA and $M = (Q, \mathcal{A}, \Delta_M, I, F)$ be an SFA.

Definition 2.20. A relation S on Q is a *simulation* on N if whenever $(p, r) \in S$,

(i) if $p \in F$, then $r \in F$ and

(ii) for all $a \in \Sigma$ and $p' \in Q$, if $p \xrightarrow{a} p'$, then there exists $r \xrightarrow{a} r'$ such that $(p', r') \in S$.

It is obvious that for any $q \in Q$, relation created by adding (q, q) to any simulation is also a simulation. That means that the reflexive closure of any simulation is also a simulation. It can also be easily shown that the transitive closure of a simulation is again a simulation and that simulation is closed under union. From this, it is obvious that there exists a unique maximal simulation that is also a preorder, unsurprisingly called the *simulation preorder*. We denote this preorder by $\preceq$ and we say that state r *simulates* state p when $p \preceq r$. We use the same definitions for the SFA M, but we replace Σ with $\mathfrak{D}_{\mathcal{A}}$.

The following well known lemma shows a link between simulation preorder and a language accepted from a state (we show the proof only for SFAs but it is very similar for FAs):

Lemma 2.21. *Let $\preceq$ be a simulation preorder on M and $p, r \in Q$. If $p \preceq r$, then $\mathcal{L}(p) \subseteq \mathcal{L}(r)$.*

Proof. The proof is by induction on the length $|w|$ of the word $w \in \mathfrak{D}_{\mathcal{A}}^*$. For word w we want to prove this statement:

$$\forall p, r \in Q.((p \preceq r \wedge w \in \mathcal{L}(p)) \implies w \in \mathcal{L}(r))$$

Base case: $|w| = 0$. In this case w is the empty word and only final states accept the empty word. From condition (i) in Definition 2.20 of simulation stems the fact that if $p \in F$ and $p \preceq r$, r is also a final state. Then $w \in \mathcal{L}(r)$.
Induction step: As the induction hypothesis (IH) assume that the statement we want to prove holds for w. Let $a \in \mathfrak{D}_{\mathcal{A}}$. We prove that the statement holds for aw. Fix $p, r \in Q$ such that:

$$\begin{aligned} p \preceq r \wedge aw \in \mathcal{L}(p) &\implies \forall b \in \mathfrak{D}_{\mathcal{A}}, p' \in Q.\left(p \xrightarrow{b} p' \implies \exists r' \in Q.(r \xrightarrow{b} r' \wedge p' \preceq r')\right) \\ &\quad \wedge \exists q \in Q.\left(p \xrightarrow{a} q \wedge w \in \mathcal{L}(q)\right) \\ &\overset{\substack{b=a,\\ p'=q}}{\implies} \exists q, r' \in Q.(r \xrightarrow{a} r' \wedge q \preceq r' \wedge w \in \mathcal{L}(q)) \\ &\overset{\text{by IH}}{\implies} \exists r' \in Q.(r \xrightarrow{a} r' \wedge w \in \mathcal{L}(r')) \\ &\implies aw \in \mathcal{L}(r) \end{aligned}$$

□

Corollary 2.22. *If $p \preceq r$ and $r \preceq p$, then $\mathcal{L}(p) = \mathcal{L}(r)$.*

Lemma 2.21 shows why the simulation preorder is useful. It can be used in solving the language inclusion problem (when we need to check whether for two automata N_1,N_2, $\mathcal{L}(N_1) \subseteq \mathcal{L}(N_2)$). Namely, as a safe but incomplete detection that the inclusion holds. The test is incomplete since the converse of Lemma 2.21 does not hold, but because simulation is polynomial in contrast to PSPACE-complete language inclusion a preliminary simulation test may pay off [1]. Another application of simulation preorder is in the state reduction of automata. Simulation-based state reduction was thoroughly investigated in [4]. They introduced an algorithm for reducing the number of states of Kripke structures using simulation preorder which can be easily adapted for FAs and SFAs. The algorithm works in three steps:

- First, merge simulation equivalent states, that is, if for states q, p, $q \preceq p$ and $p \preceq q$, merge these two states into one state q (all transition going from and to p will now go from and to q and p will be removed; let K be the new automaton after this step is done),
- then, if there is a state q, such that $q \xrightarrow{a}_K p, q \xrightarrow{a}_K r$ and $p \preceq r$, remove transition $q \xrightarrow{a}_K p$ from Δ_K and
- finally, remove all unreachable states.

That the first step preserves the language follows from Corollary 2.22 (languages accepted from simulation equivalent states are equal, so, for the language accepted by the automaton, reaching one of them is the same as reaching the other). We apply Lemma 2.21 to show that the second step also preserves the language. Let $w \in \mathcal{L}(p)$. Then $aw \in \mathcal{L}(q)$, because there is a transition $q \xrightarrow{a} p$. But because $p \preceq r$, $w \in \mathcal{L}(r)$ and from this we can also determine that $aw \in \mathcal{L}(q)$, because there is a transition $q \xrightarrow{a} r$. So the transition $q \xrightarrow{a} p$ is redundant and can be removed. Because in this step some transitions were removed, some

new states could became unreachable and we can safely remove them, thus reducing the size of the automaton even more. Important observation is that the simulation computed in the first step can be reused in the second step.

Another possible state reduction technique explored in [5] is by using simulation preorder on reverse automaton. Reverse automaton of FA N or SFA M is an automaton N^{-1} (M^{-1}) where for each transition the source and target states are swapped and the set of final states and initial states are also swapped (this automaton accepts reversed language). Using the same algorithm, we can then reduce this automaton to automaton accepting the same language. If we then reverse this automaton, we get automaton accepting the same language as original automaton.

Finally, we can also combine these two simulation preorders and merge states q, p when $q \preceq_N p$, $q \preceq_{N^{-1}} p$ and for $n \geq 1$ there does not exist a sequence of transitions $r_i \xrightarrow{a_i} s_i$ where $r_i, s_i \in Q$, $a_i \in \Sigma$ for $i \in \{0, 1, \ldots, n\}$ and $r_1 = r_n = q$ (explained in [5], later corrected in [6]). Intuitively, this last merging can be explained by realizing that every word that gets us from some initial state to q also gets us to p and every word accepted from q is also accepted from p so p encompasses q. However, combining these two methods can create a problem with preserving the preorders (if we merge two simulation equivalent states in the reverse automaton, the simulation preorder of the original automaton can change and has to be computed again). A possible way to use both simulations is to first reduce the number of states by simulation computed on original automaton and then reversing and reducing this automaton by simulation. We can repeat this until both simulations do not reduce the number of states any more.

Chapter 3

The Algorithms

In the previous chapter we gave the necessary introduction to automata theory and simulation. In this chapter we first continue by giving an overview of an existing algorithm for computing simulation preorder for FAs. Then we present the main contribution of this work, namely, three new algorithms that are modification of FA algorithm for SFAs, each offering different complexity trade-offs. In this chapter we say, for states q, p and symbol a, that q *goes above* p *via* a wrt. some relation S if there exists a transition $q \xrightarrow{a} q'$ where $(p, q') \in S$. The set of these states q' is denoted by $q^{a}\!\!\uparrow p$, so $q^{a}\!\!\uparrow p = \{\, q' \mid q \xrightarrow{a} q', (p, q') \in S \,\}$. If we say q *goes above* p *via* a wrt. $\preceq$, this means that there exists a transition $q \xrightarrow{a} q'$ where q' simulates p. All the algorithms presented in this chapter maintain an overapproximation of the simulation proeorder represented as a union $Rel \cup NotRel$ of two relations. Below, whenever we write q goes above p or $q^{a}\!\!\uparrow p$, we refer to the approximation $Rel \cup NotRel$, not to the simulation preorder itself.

3.1 FA Simulation

In this section we explain how the already existing algorithm for computing simulation preorder by Ilie, Navarro and Yu (INY) [17] works. We chose this algorithm because of its good time complexity and its relative simplicity.

It starts with some initial overapproximation of simulation preorder and then removes the pairs of states contradicting with Definition 2.20 until fixpoint, when the resulting relation becomes the simulation preorder. Algorithm SimpleINY shows the main idea (the final optimized version will then be presented as Algorithm INY). The initial relation is given by condition (i) in Definition 2.20 as $Q \times Q$ minus all the pairs of states (q, p) where q is final state and p is not (line 1). It was shown in [11] that this initial choice works only for complete automata and for non-complete automata more pairs need to be taken from the relation. This could influence the time complexity and because of this, we work only with the complete automata (this is better explained in Section 3.1.1 where we analyse the time complexity). The initial relation is called Rel and at the end of the algorithm it will be equal to the simulation preorder. Because we want process every pair removed from Rel, we save all these pairs in a set called $NotRel$. This set is initialized as the complement of Rel (line 2). The algorithm ends when all pairs in $NotRel$ are processed, that is, when $NotRel = \emptyset$.

In the next part of the algorithm, every pair of states (i, j) in $NotRel$ is processed, in two steps:

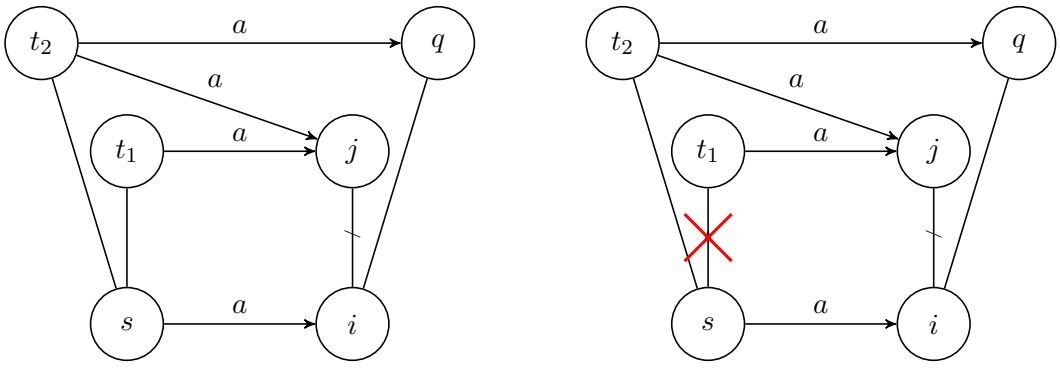
Algorithm 1: SimpleINY

Input: FA $N = (Q, \Sigma, \Delta, I, F)$
Output: $\preceq$

```
1  Rel = (Q × Q) \ (F × (Q \ F))
2  NotRel = F × (Q \ F)
3  while NotRel ≠ ∅ do
4  |   remove some (i, j) from NotRel
5  |   forall the a ∈ Σ do
6  |   |   forall the t ∈ ←Δ_a(j) do
7  |   |   |   if ∄q ∈ →Δ_a(t).((i, q) ∈ Rel ∪ NotRel) then
8  |   |   |   |   for s ∈ ←Δ_a(i) do
9  |   |   |   |   |   if (s, t) ∈ Rel then
10 |   |   |   |   |   |   Rel = Rel \ {(s, t)}
11 |   |   |   |   |   |   NotRel = NotRel ∪ {(s, t)}
12 return Rel
```

1. a check, whether condition (ii) in Definition 2.20 holds for all symbols $a \in \Sigma$ and pairs of states (s, t) where $s \in \overleftarrow{\Delta}_a(i), t \in \overleftarrow{\Delta}_a(j)$, and
2. a removal of (s, t) from Rel if the check 1 fails.

That is, we check if for all states t going via some symbol a to j, t does no longer go above i via a (because j was the last state in $t \not\curvearrowright_a i$). If t does not go above i, we can now say that each pair (s, t) where $s \in \overleftarrow{\Delta}_a(i)$ contradicts with condition (ii) in Definition 2.20 and we remove them from Rel (line 10) and add them to $NotRel$ (line 11) for future processing (if they were not removed already). We illustrate one step in Figure 3.1. Vertical lines shows which pairs are in Rel, so $(i, q), (s, t_1), (s, t_2) \in Rel$, while (i, j) is not (the pair (i, j) is being processed). Now, if we look at states t_1, t_2, which both go to j via a, we can see that t_2 goes to q, so it still goes above i but t_1 does not go above i. This means that we remove the pair (s, t_1) from Rel, because t_1 surely does not simulate s (s goes to i but t_1 does not go above i).



(a) Before (b) After

Figure 3.1: Proccessing of the pair (i, j).

To make the check on line 7 constant, we use counters. For every $i, t \in Q$ and $a \in \Sigma$, we introduce counter $N_a(i, t)$ where equality $N_a(i, t) = |t \not\curvearrowright_a i|$ should hold before and after

any pair in *NotRel* is processed. Algorithm INY uses these counters. In the beginning "every state simulates all states" ($Rel = Q \times Q$), so we initialize $N_a(i,t)$ to $|\overrightarrow{\Delta}_a(i)|$ (line 4). Then, after we start processing $(i,j) \in NotRel$, we need to update counters $N_a(t,i)$ for all $a \in \Sigma, t \in \overleftarrow{\Delta}_a(j)$. Because we removed j from $t \overset{a}{\nearrow} i$, counter $N_a(t,i)$ is decremented by one (line 11). If this counter reaches zero (line 12), this means that t no longer goes above i via a (this is the check we wanted to make faster) and we continue with updating *Rel* and *NotRel* as we did in Algorithm SimpleINY.

Algorithm 2: INY

Input: FA $N = (Q, \Sigma, \Delta, I, F)$
Output: $\preceq$

```
1  forall the q ∈ Q, a ∈ Σ do
2  |  compute Δ→_a(q), Δ←_a(q)
3  forall the q, p ∈ Q, a ∈ Σ do
4  |  N_a(q,p) = |Δ→_a(p)|
5  Rel = (Q × Q) \ (F × (Q \ F))
6  NotRel = F × (Q \ F)
7  while NotRel ≠ ∅ do
8  |  remove some (i,j) from NotRel
9  |  forall the a ∈ Σ do
10 |  |  forall the t ∈ Δ←_a(j) do
11 |  |  |  N_a(i,t) = N_a(i,t) − 1
12 |  |  |  if N_a(i,t) == 0 then
13 |  |  |  |  for s ∈ Δ←_a(i) do
14 |  |  |  |  |  if (s,t) ∈ Rel then
15 |  |  |  |  |  |  Rel = Rel \ {(s,t)}
16 |  |  |  |  |  |  NotRel = NotRel ∪ {(s,t)}
17 return Rel
```

3.1.1 Time Complexity

If n is the number of states and m is the number of transitions in FA, then the time complexity of this algorithm is $\mathcal{O}(nm)$. This is not immediately obvious, so in this section we give an informal analysis of the time complexity from [11].

The initialization on lines 1–6 is done in $\mathcal{O}(m + n^2|\Sigma| + n^2)$. As we said before, this algorithm works only with complete FAs, so for each state $q \in Q$, there are at least $|\Sigma|$ transitions going from q. Then the number of all transitions m is at least $n|\Sigma|$ and the initialization is done in $\mathcal{O}(nm)$. If we worked with non-complete automata (like they do in [11]), we would need to remove in initialization pairs (i,j) where there exists a symbol a that i does not go via into some state, but j does. This algorithm has then complexity $\mathcal{O}(\max\{nm, n^2|\Sigma|\})$.

Because in *NotRel* we save pairs of states that are not in simulation and we only save each pair at most once, line 8 is reached at most n^2. To explain the next part we need to

know the value of the sum of the initial values of all counters

$$\sum_{\substack{i,t\in Q\\ a\in\Sigma}} N_a(i,t) = \sum_{\substack{i,t\in Q\\ a\in\Sigma}} |\overrightarrow{\Delta}_a(t)|.$$

For some $t \in Q$ the sum $\sum_{a\in\Sigma} |\overrightarrow{\Delta}_a(t)|$ is equal to the number of transitions going from the state t and the sum $\sum_{t\in Q}\sum_{a\in\Sigma} |\overrightarrow{\Delta}_a(t)|$ is then equal to the number of all transitions m. Therefore,

$$\sum_{\substack{i,t\in Q\\ a\in\Sigma}} |\overrightarrow{\Delta}_a(t)| = \sum_{i\in Q}\sum_{\substack{t\in Q\\ a\in\Sigma}} |\overrightarrow{\Delta}_a(t)| = \sum_{i\in Q} m = nm.$$

Also, because counters cannot be negative (because they represent the number of states simulating some state) we can now say that line 11 (decrementing of counters) is reached at most nm times.

Now the only thing left to show is that lines 14–16 are reached at most nm times. First we need to realize that if we fix $i \in Q, a \in \Sigma$ in $N_a(i,t)$, line 13 is reached at most n times (there are n such counters). On the other hand, if we fix t, the for loop on lines 13–16 is iterated at most nm times. This stems from the similar fact as in the summation of the counters: the for loop enumerates all states in $\overleftarrow{\Delta}_a(i)$ and summed over all states i and symbols a it computes its body m times (for fixed t). If we combine these two facts, lines 14–16 are reached at most nm times.

Overall, we showed that Algorithm INY computes in $\mathcal{O}(nm)$ time.

3.2 Global SFA Simulation

In the previous section, we presented the existing algorithm for computing simulation preorders for FAs. In this section, we show how it can be generalized to SFAs. The basic idea of the generalization is the same as in [8], in which they adapted deterministic FA minimization algorithm [16] for symbolic automata. They (indirectly) use global minterm normalized form to create FA whose minimal form is the same as the minimal form of the input SFA.

The idea works like this: first we transform input SFA M to its minterm normalized form M_G. After that, interpret M_G as an FA with *Minterms* (M) as the alphabet. The next theorem shows the relation between these three automata and their simulation preorders.

Theorem 3.1. *Let $M = (Q, \mathcal{A}, \Delta_M, I, F)$ be SFA, $M_G = (Q, \mathcal{A}, \Delta_{M_G}, I, F)$ its minterm normalized form where Δ_{M_G} is given by Equation 2.1 and $N = (Q, \mathit{Minterms}(M), \Delta_{M_G}, I, F)$ be FA. Then for each $q, p \in Q$, $q \preceq_M p$ iff $q \preceq_N p$.*

Proof. We prove that $q \preceq_M p$ iff $q \preceq_{M_G} p$ and $q \preceq_{M_G} p$ iff $q \preceq_N p$, because this implies the proposition in theorem. We do this by showing that the definitions of simulation (Def. 2.20) for these automata are equivalent and because simulation preorder is a *unique maximal* simulation on $Q \times Q$, this would mean that simulation preorders are equivalent. Because the set of states Q and the set of final states F are the same for all three automata, we only need to show that the second conditions are equivalent. Because for all $a \in \mathfrak{D}_\mathcal{A}$, $q, p \in Q$, $q \xrightarrow{a}_M p$ iff $q \xrightarrow{a}_{M_G} p$, the second conditions are equivalent for M and M_G and so the equivalence $q \preceq_M p$ iff $q \preceq_{M_G} p$ holds.

Also, because for all $a \in \mathfrak{D}_\mathcal{A}$ there exists exactly one minterm φ for which $a \in [\![\varphi]\!]$. The transition $q \xrightarrow{a}_{M_G} p$ must then denote the transition $q \xrightarrow{\varphi}_{M_G} p$. Then the second condition

can be rewritten in a way that $a \in \mathfrak{D}_{\mathcal{A}}$ is replaced by $\varphi \in \mathit{Minterms}(M)$, which is exactly the same as the second condition for N and because of this, relations $\preceq_{M_G}$ and $\preceq_N$ are equal. □

Algorithm GlobalSFA then computes simulation preorder of input SFA M. Because the original algorithm works only for complete FAs, M must also be complete. We also assume that M is clean and normalized.

Algorithm 3: GlobalSFA

Input: SFA $M = (Q, \mathcal{A}, \Delta_M, I, F)$
Output: $\preceq_M$
1 compute global minterm normalized form of M, $M_G = (Q, \mathcal{A}, \Delta_{M_G}, I, F)$
2 compute $\preceq_N$ of FA $N = (Q, \mathit{Minterms}(M), \Delta_{M_G}, I, F)$
3 return $\preceq_N$

However, Algorithm GlobalSFA is plagued with one big problem: the number of minterms in $\mathit{Minterms}(M)$ is in the worst case exponential to the number of transitions in Δ_M. The next example demonstrate a worst case scenario.

Example 3.2. Let $M_k = (\{q_0, q_1, \ldots, q_{k+1}, p_1, \ldots, p_{k+1}\}, \mathbf{BDD}_k, \Delta, \{q_0\}, \{q_k, p_k\})$ (Figure 3.2) be SFA for some $k \in \mathbb{N}$. Let β_i, for some $i \in \mathbb{N}, i \leq k$ be a BDD where $[\![\beta_i]\!] = \{\, n \mid i\text{-th bit in binary representation of } n \text{ is } 1 \,\}$. This BDD and also the BDD $\neg\beta_i$ have only one node, which means these predicates are really small. However, $\mathit{Minterms}(M)$ has 2^k elements, because each minterm denotes exactly one number. For example, suppose $k = 3$, then $[\![\beta_1]\!] = \{1, 3, 5, 7\}$, $[\![\beta_2]\!] = \{2, 3, 6, 7\}$, $[\![\neg\beta_3]\!] = \{0, 1, 2, 3\}$ and thus $[\![\beta_1 \wedge \beta_2 \wedge \neg\beta_3]\!] = \{3\}$. In this example $p_i \preceq q_i$ for $i \in \{1, 2, \ldots, k+1\}$ and this automaton can be reduced to the automaton where every state p_i is removed. This example is a modification of an example from [8]. ◊

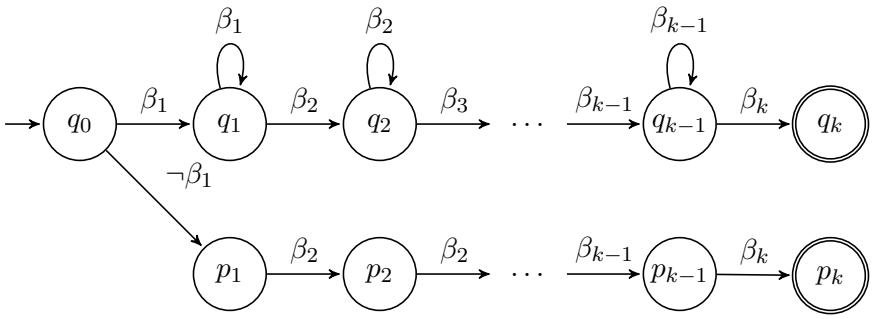


Figure 3.2: SFA M_k.

3.2.1 Implementation

We implemented this algorithm with the symbolic automata toolkit [25] implemented in C#[1]. In this toolkit, SFAs are represented as a generic class with one type parameter that determines which effective Boolean algebra is used (both $\mathbf{BDD}_k$ and $\mathbf{FOL}_{\mathfrak{D},\mathcal{T}}$ algebras

[1] source code available at https://github.com/AutomataDotNet/Automata

are implemented, the latter in the form of interface to SMT solver Z3 [10]). States are represented as integers, transition relation is implemented in the form of two dictionaries which map state q to either all transitions where q is the source state or all transitions where q is the target state.

In the implementation of Algorithm GlobalSFA we used a predicate tree introduced in Section 2.2.1 (already implemented in the tool) to compute minterms, which we then mapped to a subset of integers. After mintermization, we initialized counters and computed Δ_{M_G} (by returning on the path from minterm to the root of the predicate tree explained in Example 2.18) with minterms mapped to integers. We actually only computed "reverse" transition function implemented as a dictionary in which we mapped every state q to all transitions where q is the target state. Because we then run Algorithm INY on FA with this reverse transition function, we avoided iterating over all symbols (in this case minterms) on line 9 of Algorithm INY (lines 9 and 10 can be combined to **forall the** $t \xrightarrow{a} j$ **do**).

3.2.2 Time Complexity

Let n be the number of states of M, m be the number of transitions of M, m' the number of transition of M_G and $f(x)$ the complexity of checking the satisfiability of predicate of size x. Because M is normalized, there are at most n transitions going from a single state state and m is then bounded by n^2. As there can be at most 2^m minterms in $\mathit{Minterms}\,(M)$ and every minterm is generated from m predicates, we can compute them in $\mathcal{O}\big(2^m f(ml)\big)$ time where l is the size of the largest predicate in M. Computing Δ_{M_G} is then done in $\mathcal{O}\big(2^m f(mp) + m2^m\big)$ (every transition is replaced by transitions labelled with minterms). Because we then run standard simulation algorithm for FA N with the same number of transitions as in M_G, we can conclude that Algorithm GlobalSFA has complexity $\mathcal{O}\big(2^m f(ml) + m2^m + nm'\big)$.

It is not immediately obvious, but we can also bound m' by $m2^m$. For this we need to realise how many transitions of M_G can be labelled by the same minterm ψ. Let m_ψ be the number of these transitions. If we have minterm in the form $\psi = \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_k \wedge \neg\varphi_{k+1} \wedge \cdots \wedge \neg\varphi_m$ where, for $i = 1, 2, \ldots, m$, φ_i is some predicate labelling a transition in M, then the number of transitions $m_\psi = k$, because every transition $q_i \xrightarrow{\varphi_i}_M p_i$, for $1 \leq i \leq k$, is replaced by $q_i \xrightarrow{\psi}_{M_G} p_i$ (and also by all the other minterms that are created from φ_i). In the worst case, the number of minterms where k predicates are not negated (these minterms are created from k predicates) is $\binom{m}{k}$ (because we choose k transitions from m transitions). So the number of transitions labelled with minterms created from k predicates is $\binom{m}{k}k$ and the numbers of transitions in M_G is

$$m' = \sum_{k=0}^{m} k\binom{m}{k} = m2^{m-1} \tag{3.1}$$

and the complexity of Algorithm GlobalSFA is $\mathcal{O}\big(2^m f(ml) + nm2^m\big)$. See Appendix B for the detailed derivation of Equation 3.1.

3.3 Local SFA Simulation

In the previous section we introduced simulation algorithm for SFAs that uses minterms generated by predicates labelling transitions. As we explained, this mintermization can in worst case have exponential time complexity to the number of transitions. In this section we introduce another two algorithms that try to remedy this problem. First, we introduce

an algorithm where no mintermization and no counters are used and then we present its modification where local mintermization is used.

Algorithm 4: NoCountSFA

Input: FA $M = (Q, \mathcal{A}, \Delta, I, F)$
Output: $\preceq$

```
1  Rel = (Q × Q) \ (F × (Q \ F))
2  NotRel = F × (Q \ F)
3  while NotRel ≠ ∅ do
4      remove some (i, j) from NotRel
5      forall the t --φ_tj--> j do
6          ψ = φ_tj
7          forall the t --φ_tk--> k do
8              if (i, k) ∈ Rel ∪ NotRel then
9                  ψ = ψ ∧ ¬φ_tk
10         forall the s --φ_si--> i do
11             if (s, t) ∈ Rel then
12                 if IsSat(ψ ∧ φ_si) then
                       // [[ψ]] ∪ [[φ_si]] ≠ ∅
13                     Rel = Rel \ {(s, t)}
14                     NotRel = NotRel ∪ {(s, t)}
15 return Rel
```

Algorithm NoCountSFA is a modification of Algorithm SimpleINY adapted to the symbolic setting. Initialization is done in the same way as in Algorithm SimpleINY, the initial relation is given by condition (i) of Definition 2.20. Again, *Rel* is a relation on Q which in the end of computation will be equal to the simulation preorder, and *NotRel* is the set of all pairs that are taken from *Rel* and then, later, processed. This processing is slightly different than in Algorithm SimpleINY. Again, during the processing of pair (i, j) we want to remove pairs (s, t) where s goes to i and t goes to j that are contradicting condition (ii) in Definition 2.20. In Algorithm SimpleINY we looked for all such states t where t did not go above i via some symbol. However, because we are now working with SFAs, we would like to find all symbols (a predicate denoting this symbols) $a \in \mathfrak{D}_{\mathcal{A}}$ where t goes to j via a but not above t via a. This is done on lines 6–9 where ψ is the predicate that denotes all such symbols. First, it is initialized to φ_{tj}, because φ_{tj} is the predicate denoting all symbols $a \in \mathfrak{D}_{\mathcal{A}}$ that t goes via a to j (we assume that M is normalized). Then we remove all symbols $b \in \mathfrak{D}_{\mathcal{A}}$ from ψ where t goes above i via b (lines 7–9). After that, we check, for each state s that goes to i, if it goes to i via some symbol in $[\![\psi]\!]$ (line 12). If it does, we remove the pair (s, t) from *Rel* and save it in *NotRel* for later processing. After all pairs in *NotRel* are processed, *Rel* becomes the simulation preorder.

Notice that because we work with symbolic representation, we now enumerate all states s going to i on line 10 and not only those that go to i via some symbol from $[\![\psi]\!]$, because we do not know beforehand if s goes to i via some symbol from $[\![\psi]\!]$. This is in contrast with Algorithm SimpleINY where we needed to only enumerate those states where there was a transition going from them to i via some concrete symbol. Because of this, the time complexity is worse.

Algorithm 5: LocalSFA

Input: FA $M = (Q, \mathcal{A}, \Delta_M, I, F)$
Output: $\preceq$

1 compute local minterm normalized form of M, $M_L = (Q, \mathcal{A}, \Delta_{M_L}, I, F)$
2 **forall the** $q \in Q$ **do**
3 **forall the** $\psi \in \mathit{Minterms}(q)$ **do**
4 **forall the** $p \in Q$ **do**
5 $N_\psi(q,p) = |\{\, r \mid q \xrightarrow{\psi}_{M_L} r \,\}|$
6 $\mathit{Rel} = (Q \times Q) \setminus (F \times (Q \setminus F))$
7 $\mathit{NotRel} = F \times (Q \setminus F)$
8 **while** $\mathit{NotRel} \neq \emptyset$ **do**
9 remove some (i,j) from NotRel
10 **forall the** $t \xrightarrow{\psi_{tj}}_{M_L} j$ **do**
11 $N_{\psi_{tj}}(i,t) = N_{\psi_{tj}}(i,t) - 1$
12 **if** $N_{\psi_{tj}}(i,t) == 0$ **then**
13 **for** $s \xrightarrow{\varphi_{si}}_{M} i$ **do**
14 **if** $(s,t) \in \mathit{Rel}$ *and* $\mathit{IsSat}(\psi_{tj} \wedge \varphi_{si})$ **then**
15 $\mathit{Rel} = \mathit{Rel} \setminus \{(s,t)\}$
16 $\mathit{NotRel} = \mathit{NotRel} \cup \{(s,t)\}$
17 return Rel

In Section 3.3.2 we will explain that the for loop on lines 7–9 of Algorithm NoCountSFA introduces a factor $n \sum_{q \in Q} m_q^2$ to the time complexity, where n is the number of states and m_q the number of transitions with source state q. Now we show how to replace the for loop with more efficient test using counters. We cannot have counters $N_a(t,i)$ for some $a \in \mathfrak{D}_\mathcal{A}, t, i \in Q$ as we have in Algorithm INY, because the domain $\mathfrak{D}_\mathcal{A}$ could be infinite and even if it was not, we would need to constantly check if a was in the set of symbols denoted by some predicate. This is why we want to have counters $N_\varphi(t,i)$ for some predicate $\varphi \in \Psi$. We cannot use predicates that label transitions because the sets of symbols these predicates denote can intersect. This is where the local minterm normalized form comes into play. Because SFAs in this form have the property that for each $q \in Q, a \in \mathfrak{D}_\mathcal{A}$ there exists exactly one $\psi \in \mathit{Minterms}(q)$ where $a \in [\![\psi]\!]$ and all transitions going from state q are labelled by predicates from $\mathit{Minterms}(q)$, we can say that for $p \in Q, a, b \in [\![\psi]\!]$, $q\pitchfork p = q\pitchfork b$. Let $q\pitchfork i$ denote the set of states r where $q \xrightarrow{\psi} r$ and $i \preceq r$. This means that we can have counter $N_\psi(q,i)$ that will be equal to $|q\pitchfork i|$ except during the processing of pair from NotRel (similarly to counters in Algorithm INY). However, using predicates for counters is not the only difference from counters in Algorithm INY. Because each $q \in Q$ has different set $\mathit{Minterms}(q)$, this means, that while there is a counter $N_\psi(q,p)$ for some $p \in Q, \psi \in \mathit{Minterms}(q)$, the counter $N_\psi(i,j)$ for some $i, j \in Q$ might not make a sense, because generally $\psi \notin \mathit{Minterms}(i)$. This means that we have counters $N_\psi(q,p)$ for all $q, p \in Q$, $\psi \in \mathit{Minterms}(q)$.

Algorithm LocalSFA uses local mintermizations and counters. The initialization is done in the same way as in Algorithm INY, $N_\psi(q,p) = |\{\, r \mid q \xrightarrow{\psi}_{M_L} r \,\}|$ (line 5). Then, during the processing of the pair (i,j), we decrement all counters $N_{\psi_{tj}}(t,i)$ where $t \xrightarrow{\psi_{tj}}_{M_L} j$ (because j was taken from the set $t^{\psi_{tj}}\!\!\pitchfork i$). If the counter reaches zero, we want to remove all pairs (s,t)

where $s \overset{\varphi_{si}}{\to}_M i$ and there exists some $a \in [\![\varphi_{si}]\!]$ and at the same time $a \in [\![\psi_{tj}]\!]$, so we check the satisfiability of $\psi_{tj} \wedge \varphi_{si}$ (line 14). If such symbol exists, we can be sure that t does not simulate s and so we remove the pair (s, t) from *Rel*.

As with Algorithm INY, Algorithms NoCountSFA and LocalSFA works only with complete SFAs. We also assume that M is clean and normalized.

3.3.1 Implementation

We implemented both Algorithms NoCountSFA and LocalSFA in the symbolic automata toolkit [25]. To make Algorithm NoCountSFA faster, we process pairs (i, j) in *NotRel* with the same lower state at the same time. Lower state is the state i in the pair (i, j). This means that if there are pairs $(i, j_1), (i, j_2)$ in *NotRel*, we take the set of all states t going to either j_1 or j_2 and then we create the disjuction $\psi = \bigvee \varphi_{tk}$ for all φ_{tk} where exists $t \overset{\varphi_{tk}}{\to} k$ such that $(i, k) \in Rel$. The predicate ψ then denotes all the symbols a such that t goes above i via a. The check on line 12 of Algorithm NoCountSFA is then for satisfiability of $\neg\psi \wedge \varphi_{si}$. In this example we saved one iteration of the for loop on line 10 because we processed pairs $(i, j_1), (i, j_2)$ at the same time. The effect of the optimization is the more significant the more pairs with the same lower state appear in *NotRel*. To implement this optimization efficiently, we divide *NotRel* to sets $NotRel_i$ for all $i \in Q$ where $NotRel_i$ is the set of all $j \in Q$ where (i, j) would be in *NotRel*. We then save all those i for which $NotRel_i \neq \emptyset$ for later processing (instead of saving pairs).

The local mintermization is implemented similarly as global mintermization in Section 3.2.1. Counters are now implemented as 2D arrays of arrays of integers with variable sizes (the size depends on how much minterms were created from a state). In this situation we can also implement an optimization: when a counter $N_{\psi_1}(i, t)$ reaches zero, we save ψ_1 for a pair (i, t) but we do not immediately start with removing of pairs. We first process all pairs in *NotRel* and if some other counter $N_{\psi_2}(i, t)$ reaches zero, we now save $\psi_1 \vee \psi_2$ for a pair (i, t), etc. After processing all pairs in *NotRel*, we check if we have saved for any pair of states (i, t) this predicate and if yes, we continue with this predicate instead of predicate ψ_{tj} in the for loop on line 13 of Algorithm LocalSFA.

3.3.2 Time Complexity

The time complexity of Algorithm NoCountSFA is $\mathcal{O}\big(n \sum_{q \in Q} m_q^2 + m^2 f(m_p k)\big)$ where n is the number of states, m is the number of transitions, m_q is the number of transitions with the source state q, p is the state with the most number of transitions going from it, k is the size of the largest predicate in M and $f(x)$ is the complexity of checking the satisfiability of predicate of the size x. We now examine why. Initialization (lines 1 and 2) is obviously done in $\mathcal{O}\big(n^2\big)$. Line 4 is also reached at most n^2 times, because we can save each pair of states in *NotRel* only once. Line 6 is for fixed i reached at most m times (because we enumerate all transitions going to j and summed over all $j \in Q$ this is all transitions in M), so for all $i \in Q$ it is reached at most nm times. Let $i, j, t \in Q$ be fixed such that t goes to j. Lines 8–9 are then reached m_t times. There are m_t states that t goes to (M is normalized), so summed over all such j these lines are reached at most m_t^2 times. This means that summed over all $i, t \in Q$, these lines are reached at most $n \sum_{t \in Q} m_t^2$ times. The last thing to show is that lines 12–14 are reached at most m^2 times. If we fix t and j, these lines are reached at most m times (again we enumerate all transitions going to some state i and summed over all states this is all transitions). Because there is only one transition

going from t to j, summing over all such states t, j where there is a transitions between them, we can conclude that these lines are reached at most m^2. Also the predicate ψ is a conjunction of m_t predicates and $n^2 \leq nm \leq n \sum_{q \in Q} m_q^2$ so Algorithm NoCountSFA has the time complexity

$$\mathcal{O}\big(n \sum_{q \in Q} m_q^2 + m^2 f(m_p k)\big).$$

We now examine the time complexity of Algorithm LocalSFA. Let for some $q \in Q$, r_q be the number of minterms in $\mathit{Minterms}\,(q)$. Using the same reasoning as for global mintermization, M_L can be computed in $\mathcal{O}\big(\sum_{q \in Q}(r_q f(m_q k) + m_q r_q)\big)$. Let r be the number of all local minterms, that is $r = \sum_{q \in Q} r_q$. The initialization on lines 2–5 is done in $\mathcal{O}\big(nr\big)$ time ($|\{\, r \mid q \xrightarrow{\psi}_{M_L} r \,\}|$ is computed during mintermization). Let m' be the number of transitions in M_L. Using the same reasoning as in FA simulation, we can say that the summation of all counters at the beginning is nm' and so line 11 is reached at most nm' times. For fixed i, line 12 is reached r times, because there are r counters $N_{\psi_{tj}}(i, t)$ that can reach zero only once (for each $t \in Q$ there is one counter $N_\psi(i, t)$ for each $\psi \in \mathit{Minterms}\,(t)$). If we now fix t and ψ_{tj}, lines 13–16 are reached at most m times. All in all, these lines are reached at most rm times. Because ψ_{tj} is minterm created from m_t transitions and $r \leq m'$, the time complexity of this algorithm is $\mathcal{O}\big(\sum_{q \in Q}(r_q f(m_q k) + m_q r_q) + nm' + mr f(m_p k)\big)$. Because for some $q \in Q$, r_q is bounded by 2^{m_q} (because r_q is the number of minterms) and m' is bounded by $\sum_{q \in Q} m_q 2^{m_q}$ (by Equation 3.1), the final time complexity of Algorithm LocalSFA is

$$\mathcal{O}\big(\sum_{q \in Q}(f(m_q k) 2^{m_q} + m_q 2^{m_q}) + n \sum_{q \in Q} m_q 2^{m_q} + m f(m_p k) \sum_{q \in Q} 2^{m_q}\big).$$

Comparing the complexities of the algorithms, it is obvious that the time complexity of Algorithm LocalSFA is worse than the time complexity of Algorithm NoCountSFA. Local mintermization adds the first factor to the time complexity of Algorithm LocalSFA, in the second factor one m_q is replaced with 2^{m_q} and in the third factor one m is replaced with $\sum_{q \in Q} 2^{m_q}$. However, the number of created minterms can be really small compared to the number of transitions, especially if m_q is small for all $q \in Q$ and predicates labelling transitions do not intersect much. This means that Algorithm LocalSFA can sometimes outperform Algorithm NoCountSFA.

Chapter 4

Evaluation

In the last chapter we introduced three new algorithms for computing simulation preorder for SFAs. In this chapter we present experimental evaluation of these algorithm implemented with the symbolic automata toolkit [25] (we talked about implementation details in Sections 3.2.1 and 3.3.1). All the experiments were run on Intel Core i5–3230M CPU 2.6 GHZ with 4 GB of RAM. The results of experiments can be found on the attached CD (see Appendix A for a list).

4.1 Regular Expressions Experiments

In this section we evaluate the algorithms on SFAs created from 1920 regular expressions over the UTF-16 alphabet (we use the algebra $\mathbf{BDD}_{16}$). These regular expressions were taken from the website [23] which contains a library of regular expressions contributed by people all over the world, created for different purposes, such as detecting emails, URIs, dates, times, addresses, phone numbers, etc. SFAs created from these regular expressions were used during evaluation of algorithms computing minimal SFAs from deterministic SFAs [8] and during the evaluation of bisimulation algorithms for SFAs [9]. The largest automaton has 3190 states and 10 702 transitions and these SFAs have on average 2.5 transitions per state. Because UTF-16 alphabet is too large, only symbolic representation is feasible for these automata.

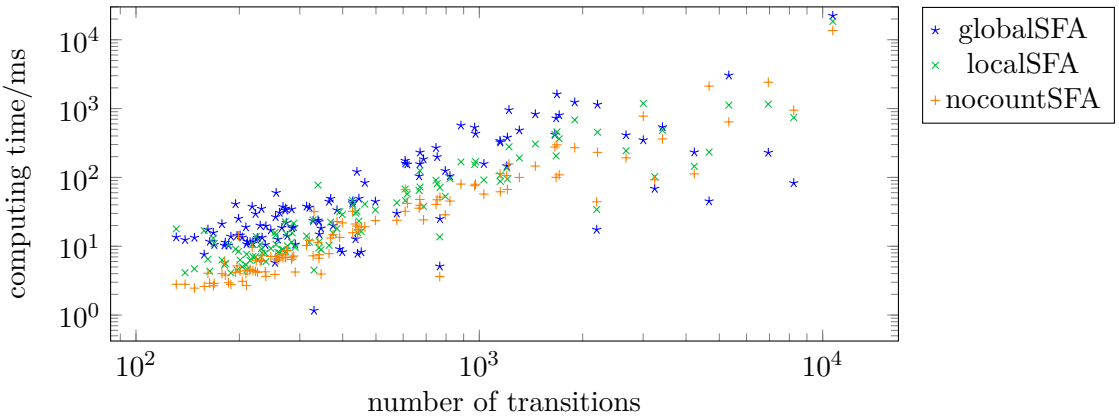


Figure 4.1: Runtimes of algorithms for computing simulation preorder on SFAs created from reg. expressions.

First, we compared the runtimes of Algorithms GlobalSFA, NoCountSFA and LocalSFA. Figure 4.1 illustrates the runtimes where on x-axis is the number of transitions of automaton and on the y-axis the time it took to compute algorithms in milliseconds. Both scales are logarithmic and the figure does not show runtimes where all three algorithms took less than 10 ms to finish. In 96.5 % cases Algorithms NoCountSFA and LocalSFA had similar runtimes (differing less than 10 ms). In 58 cases Algorithm NoCountSFA performed better (on average by 0.1 s with the largest difference 4.8 s) and in 7 cases Algorithm LocalSFA performed better (on average by 0.5 s with the largest difference 1.8 s). Comparing Algorithms NoCountSFA and GlobalSFA, these numbers were as follow: in 94 % cases they had similar runtimes, in 98 cases Algorithm NoCountSFA was better and in 14 cases Algorithm GlobalSFA was better (the average differences were 0.2 s and 0.4 s and the largest differences were 8.6 s and 2.1 s respectively). For Algorithms LocalSFA and GlobalSFA the numbers were: 97 % of similar cases, in 71 cases Algorithm LocalSFA was better and in 19 cases Algorithm GlobalSFA was better (on average by 0.2 s and 0.1 s with largest differences 4 s and 1.2 s).

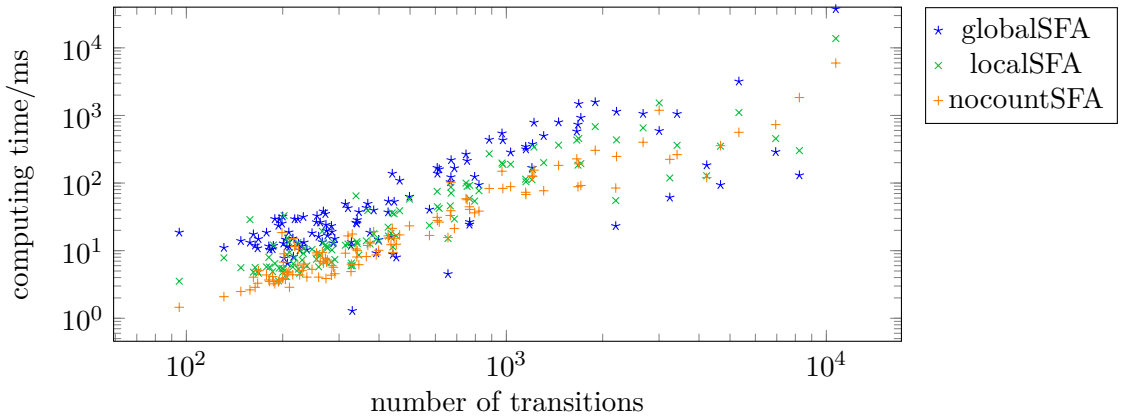


Figure 4.2: Runtimes of algorithms for computing simulation preorder on reverse SFAs created from reg. expressions.

We also compared runtimes of algorithms on reversed SFAs (Figure 4.2). In 96 % cases Algorithms NoCountSFA and LocalSFA had similar runtimes, Algorithm NoCountSFA preformed better 52 times (on average by 0.2 s with the largest difference 7.7 s) while Algorithm LocalSFA 9 times (on average by 0.9 s with the largest difference 1.5 s). In the case of Algorithms NoCountSFA and GlobalSFA the numbers were as follow: 94 % of similar cases, 101/14 cases of better performance on average by 0.4 s/0.4 s with largest differences 31.5 s/1.7 s. Finally, Algorithms LocalSFA and GlobalSFA they were: 94 % of similar cases, 89/23 cases of better performance on average by 0.4 s/0.1 s with largest differences 23.7 s/0.9 s.

To conclude, in most cases Algorithm NoCountSFA had the best performance. However, in some cases Algorithm GlobalSFA outperformed other two, sometimes by quite big margin. This can be explained by the small number of distinct predicates occurring in these automata. For this reason, the automata generated relatively small number of global minterms (sometimes smaller than the number of transitions), so the exponentional blow-up in Algorithm GlobalSFA did not occur. In Section 4.2 we show situations where this algorithm fails miserably. Interestingly, Algorithm LocalSFA usually performed worse than one of those two, but better than another. This means that it "mimicked" the behaviour

of the better one: if Algorithm NoCountSFA was better than Algorithm GlobalSFA, so was Algorithm LocalSFA. Conversely, if Algorithm GlobalSFA was better than Algorithm NoCountSFA, Algorithm LocalSFA was also better than Algorithm NoCountSFA.

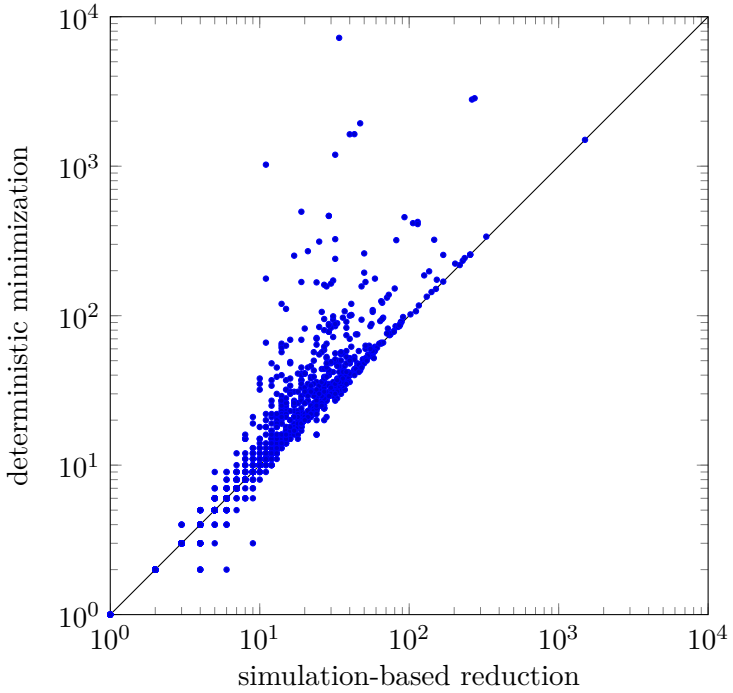


Figure 4.3: Comparison of the number of states using simulation-based reduction and deterministic minimization on SFAs created from reg. expressions

In the next experiment we compared simulation-based state reduction with deterministic minimization. For reduction by simulation we used the technique introduced at the end of Section 2.3. This means that we first reduced automaton with simulation preorder, by merging simulation equivalent states, deleting unneeded transitions and removing unreachable states. Then we reversed the reduced automaton. After that we reduced the reversed automaton with simulation and again reversed the result. We repeated this process until a fixpoint, where reduction by simulation did not result in smaller automaton. Because Algorithm NoCountSFA outperformed the other two in most cases, we used this algorithm for computing simulation. For deterministic minimization we first determinized the automaton and then used the algorithm introduced in [8] for minimization. In 11 cases the determinization timed out after 100 seconds. Figure 4.3 shows the relation between the number of states after the simulation-based reduction with the deterministic minimization (logarithmic scales). In 64 % cases the automata did not differ in size while in 28.3 % cases the simulation-based reduction created smaller automata (on average by 58 states). In other cases the deterministic minimal automaton was smaller than the one created by simulation by up to 8 states. On average, the simulation-based method reduced SFAs by 31 %. Deterministic minimization has on average created bigger automata by 4 %, because in some cases after determinization the resulting automata were much bigger than original. All in all, the simulation-based reduction created from some SFAs much smaller automata than deterministic minimization.

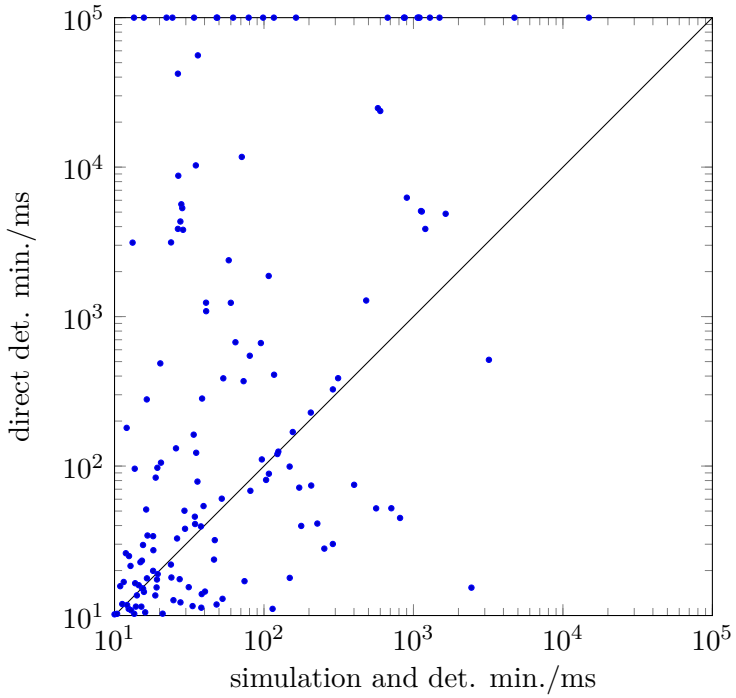


Figure 4.4: Comparison of runtimes of simulation-based reduction followed deterministic minimization and direct deterministic minimization on SFAs created from reg. expressions

Because simulation-reduced automaton is smaller but still non-deterministic automaton, we also wanted to check whether the runtime of simulation followed by deterministic minimization is faster than deterministic minimization of original automaton. Figure 4.4 shows this comparison, where we compare runtimes of direct minimization and simulation followed by deterministic minimization. The runtimes are in milliseconds and maximal value means that the determinization timed out. There were four cases where determinization on reduced SFAs timed out after 100 seconds. In 91 cases direct deterministic minimization was faster with the biggest difference 2.6 s and average difference 0.1 s. In 114 cases deterministic minimization after simulation reduction was faster with the biggest difference 55.8 s and average difference 2.6 s. In other cases the difference was less than 10 ms. Clearly, simulation reduction often significantly improves efficiency of determinization, especially in hard cases. The differences in determinization time are usually either vast or negligeable

4.2 Edge Case Experiments

To show the problematic case for which the Algorithm GlobalSFA fails, we run all three algorithms on SFA M_k from the Example 3.2 for $1 < k < 20$. Figure 4.5 compares runtimes of these algorithms, where x-axis is the value of k and y-axis is the computation time in ms. It also shows that while Algorithms NoCountSFA and LocalSFA take roughly the same time (1 ms) to compute the simulation preorder for all k, for Algorithm GlobalSFA the execution time grows exponentially and for $k > 17$ it runs out of memory. Obviously, as this example was created as an edge case for Algorithm GlobalSFA, it does not always reflect the real situation. Also, because there are at most two transitions going from each state, the local mintermization in Algorithm LocalSFA had small impact on the runtime

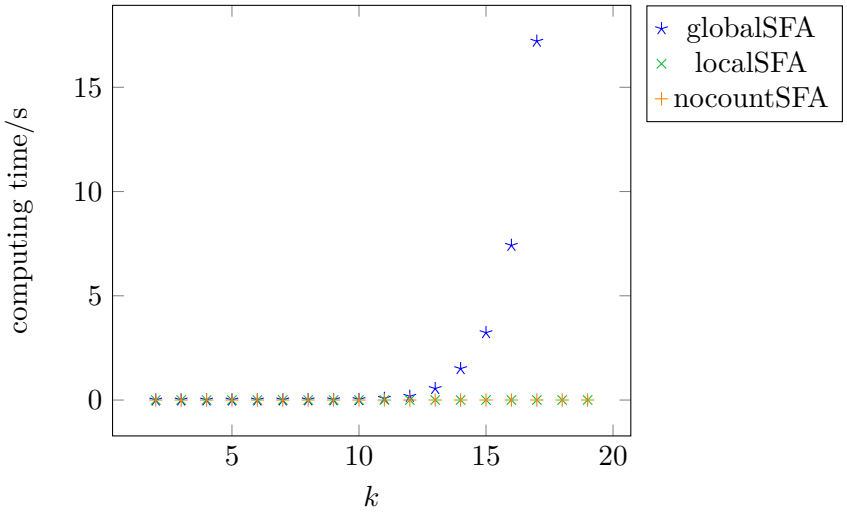


Figure 4.5: Runtimes of algorithms on M_k

of algorithm. In the next experiment we show the situation where both global and local mintermizations cause problems.

4.3 WS1S Experiments

For experiments in this sections we used SFAs created during the decision procedure for *weak-monadic second order logic of one successor* (WS1S). This logic uses variables that can represent both numbers and finite sets of numbers and it has many applications such as in software verification or computational linguistic. The WS1S decision procedure creates an automaton from a WS1S formula whose language is the set of all encoded assignments of variables in the predicate such that it satisfies the formula. See [7] for more information. We used two batches of SFAs: 95 deterministic ones from the tool VATA [19] and 38 non-deterministic from dWiNA [13]. These automata had at most 2508 states and 34 374 transitions and, on average, 6 transitions per state. They use the algebra $\mathbf{BDD}_k$ for suitably chosen k.

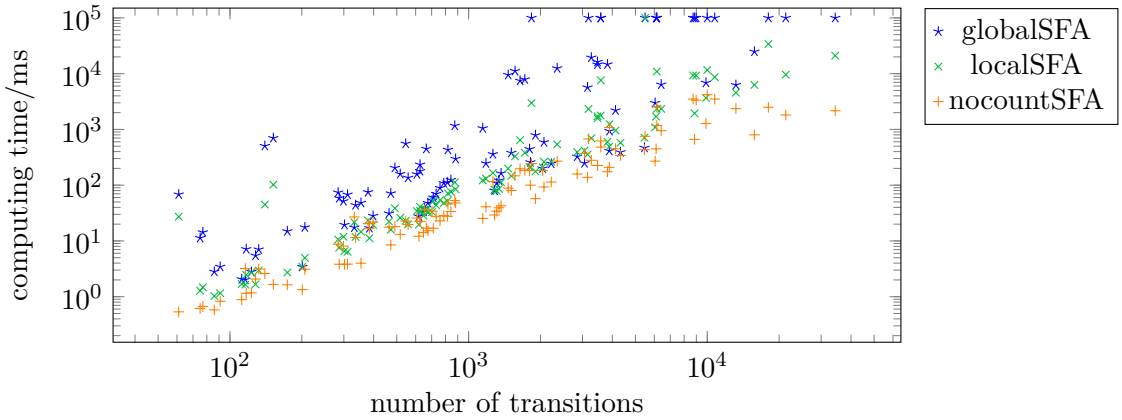


Figure 4.6: Runtimes of algorithms for computing simulation preorder on SFAs created during the WS1S decision procedure.

We again compare runtimes of simulation algorithms for original and also reversed automata. Even though simulation is usually used for non-deterministic automata, we included the results from deterministic SFAs, because the algorithms behaviours were similar as for non-deterministic ones. Figure 4.6 shows runtimes on original SFAs. Again, we removed SFAs where each algorithm took less than 10 ms to compute simulation preorder. Algorithm GlobalSFA has run out of memory in 18 cases and there was one case where Algorithm LocalSFA has run out of memory. Algorithm NoCountSFA had similar runtime as Algorithm LocalSFA in 56 cases and in 76 cases it outperformed it (on average by 1.8 s). Algorithm NoCountSFA also outperformed (77 cases) or had similar runtime (36 cases) as Algorithm GlobalSFA. There were only 2 cases where Algorithm GlobalSFA was better than NoCountSFA, and only by 0.1 s. Comparison of Algorithms LocalSFA and GlobalSFA was similar; in 41 cases they performed similarly, in 66 cases Algorithm LocalSFA outperformed GlobalSFA on average by 2.5 s and in 8 cases Algorithm GlobalSFA was better (by up to 0.3 s).

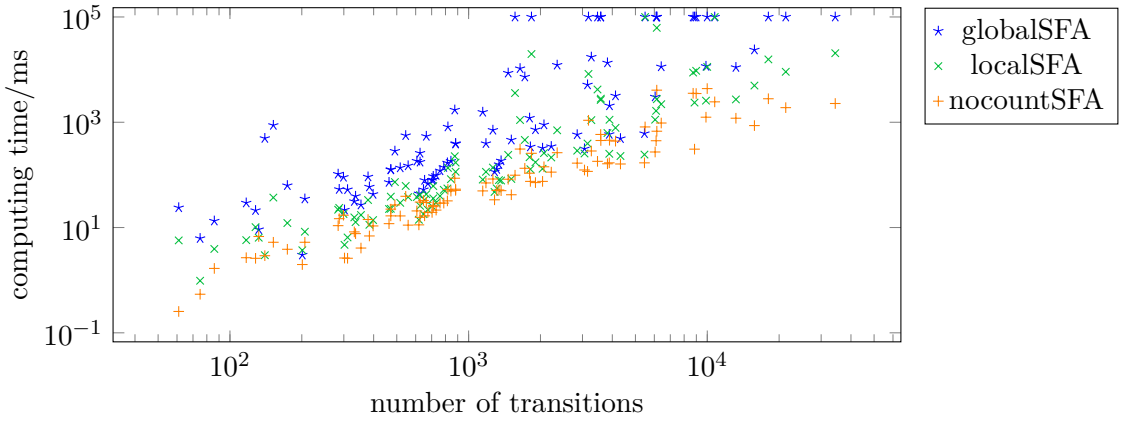
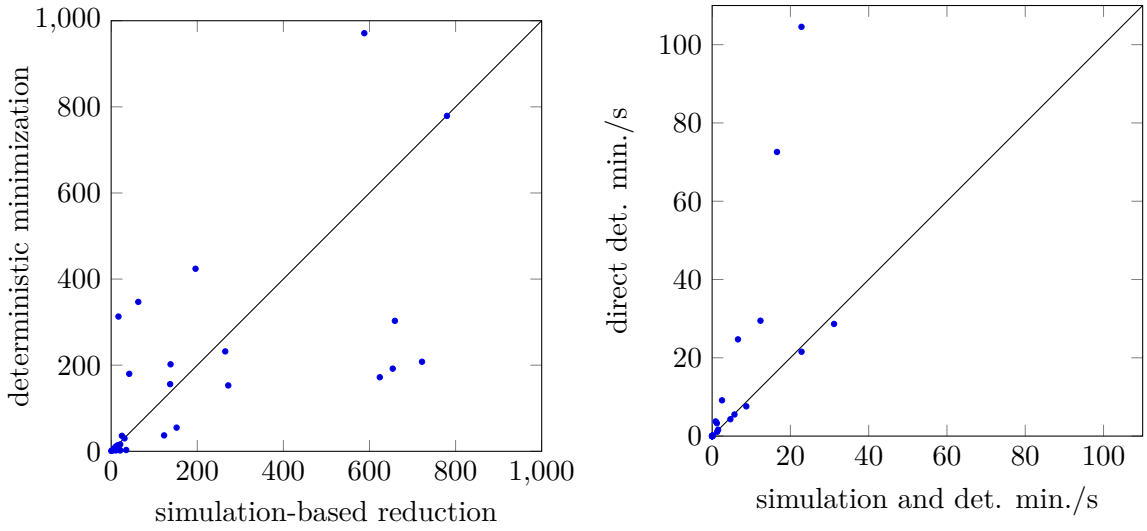


Figure 4.7: Runtimes of algorithms for computing simulation preorder on reverse SFAs created during the WS1S decision procedure.

Figure 4.7 shows the runtimes of algorithms on reversed SFAs. Now, Algorithms GlobalSFA and LocalSFA run out of memory in 21 and 3 cases respectively. Also, both Algorithms NoCountSFA and LocalSFA were similar to (in 29 and 32 cases respectively) or outperformed (in 83 and 80 cases respectively; on average by 1.9 s) Algorithm GlobalSFA. Algorithms NoCountSFA and LocalSFA performed similarly in 58 cases and in 71 cases Algorithm NoCountSFA performed better, on average by 2.3 s. Obviously, for these automata using global mintermization is out of question and using Algorithm NoCountSFA is the best course of action.

We again compared the simulation-based state reduction with deterministic minimization, but only for non-deterministic automata. Figure 4.8a compares the number of states after deterministic minimization and simulation reduction and Figure 4.8b compares the runtimes of direct deterministic minimization and simulation followed by deterministic minimization run on reduced automaton. In four cases both determinizations timed out, while there was one case where only direct determinization timed out. In 18 cases deterministic minimization produced smaller automata, while in 8 cases simulation reduction produced smaller automata. On average, simulation reduced the number of states by 30 %, with maximal reduction of 72 % (from 2217 states to 624 states). Deterministic minimization created on average bigger automata by 20 %, because in some cases blow-up of states oc-



(a) State reduction comparison

(b) Runtime comparison

Figure 4.8: Comparison of the number of states using simulation-based reduction and det. min. and runtimes of simulation-based reduction followed by det. min. and direct det. min. on SFAs created during the WS1S decision procedure

curred during determinization. Direct minimization was faster in 7 cases by up to 2.50 s (on average by 0.8 s). Minimization with simulation was faster in 10 cases by up to 81 s (18 s on average). While simulation did not usually create smaller automata than deterministic minimization, it is obvious that using simulation-based reduction before determinization can vastly improve the performance of determinization.

In this chapter we showed that Algorithm NoCountSFA is in most cases the best choice for computing simulation preorder on SFAs. We also demonstrated that using simulation before determinization to directly reduce the number of states of nondeterministic automaton can greatly improve efficiency of determinization.

Chapter 5

Conclusion

In this work we introduced three new algorithms for computing simulation preorder on SFAs. First one, Algorithm GlobalSFA, was based on global mintermization. Second one, Algorithm NoCountSFA, did not use mintermization but only the capabilities of symbolic automata. The last one, Algorithm LocalSFA was a modification of Algorithm NoCountSFA that used local mintermization. We implemented and evaluated them on SFAs created from regular expressions and during the decision procedure of WS1S logic. The two of them that were based on mintermization did not scale well with larger automata while Algorithm NoCountSFA had no problem with them. We also compared simulation-based reduction with deterministic minimization where on benchmark of SFAs created from regular expressions simulation-reduction usually created smaller automata than deterministic minimization. On benchmarks of SFAs created during the decision procedure of WS1S logic in some cases simulation-based reduction was better and in another deterministic minimization. Finally, we showed that using simulation to reduce the number of states before determinization can vastly improve the runtime of determinization of SFAs.

The next step for this work is to come up with an algorithm that improves efficiency by working with equivalence classes of the current approximation of simulation instead of with individual states. This technique was used in [22]. Another possible direction is extending algorithms presented in this work to the symbolic encoding of automata similar to that of MONA [12] where predicate constraint also the target states, not only input symbols.

Bibliography

[1] Abdulla, P. A.; Chen, Y.; Holík, L.; et al.: When Simulation Meets Antichains. In *TACAS*, *Lecture Notes in Computer Science*, vol. 6015. Springer. 2010. pp. 158–174.

[2] Bradley, A. R.; Manna, Z.: *The Calculus of Computation: Decision Procedures with Applications to Verification*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.. 2007. ISBN 3540741127.

[3] Bryant, R. E.: Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*. vol. 35, no. 8. August 1986: pp. 677–691. ISSN 0018-9340. doi:10.1109/TC.1986.1676819.

[4] Bustan, D.; Grumberg, O.: Simulation-based Minimization. *ACM Trans. Comput. Logic*. vol. 4, no. 2. April 2003: pp. 181–206. ISSN 1529-3785. doi:10.1145/635499.635502.

[5] Champarnaud, J.-M.; Coulon, F.: NFA Reduction Algorithms by Means of Regular Inequalities. *Theor. Comput. Sci.*. vol. 327, no. 3. November 2004: pp. 241–253. ISSN 0304-3975. doi:10.1016/j.tcs.2004.02.048.

[6] Champarnaud, J.-M.; Coulon, F.: Erratum to "NFA Reduction Algorithms by Means of Regular Inequalities" [Theoret. Comput. Sci. 327(2004) 241-253]. *Theor. Comput. Sci.*. vol. 347, no. 1-2. November 2005: pp. 437–440. ISSN 0304-3975. doi:10.1016/j.tcs.2005.07.001.

[7] Comon, H.; Dauchet, M.; Gilleron, R.; et al.: Tree Automata Techniques and Applications. 2007. [Online; accessed 13. 05. 2017]. Retrieved from: http://www.grappa.univ-lille3.fr/tata

[8] D'Antoni, L.; Veanes, M.: Minimization of Symbolic Automata. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: ACM. 2014. ISBN 978-1-4503-2544-8. pp. 541–553. doi:10.1145/2535838.2535849.

[9] D'Antoni, L.; Veanes, M.: Forward Bisimulations for Nondeterministic Symbolic Finite Automata. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'2017. Berlin, Heidelberg: Springer-Verlag. 2017. ISBN 978-3-662-54577-5. pp. 518–534. doi:10.1007/978-3-662-54577-5_30.

[10] De Moura, L.; Bjørner, N.: Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for*

the Construction and Analysis of Systems. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag. 2008. ISBN 3-540-78799-2, 978-3-540-78799-0. pp. 337–340.

[11] Eberl, M.: *Efficient and Verified Computation of Simulation Relations on NFAs*. Bachelor's thesis. Technical University of Munich. Munich. 2012.

[12] Elgaard, J.; Klarlund, N.; Møller, A.: MONA 1.X: New Techniques for WS1S and WS2S. In *Proceedings of the 10th International Conference on Computer Aided Verification*. CAV '98. London, UK, UK: Springer-Verlag. 1998. ISBN 3-540-64608-6. pp. 516–520.

[13] Fiedor, T.; Holík, L.; Lengál, O.; et al.: Nested Antichains for WS1S. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. New York, NY, USA: Springer-Verlag New York, Inc.. 2015. ISBN 978-3-662-46680-3. pp. 658–674. doi:10.1007/978-3-662-46681-0_59.

[14] Henzinger, M. R.; Henzinger, T. A.; Kopke, P. W.: Computing Simulations on Finite and Infinite Graphs. In *FOCS*. IEEE Computer Society. 1995. pp. 453–462.

[15] Hooimeijer, P.; Veanes, M.: An Evaluation of Automata Algorithms for String Analysis. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'11. Berlin, Heidelberg: Springer-Verlag. 2011. ISBN 978-3-642-18274-7. pp. 248–262.

[16] Hopcroft, J. E.: An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. Technical report. Stanford University. Stanford, CA, USA. 1971.

[17] Ilie, L.; Navarro, G.; Yu, S.: On NFA Reductions. In *Theory Is Forever*, *Lecture Notes in Computer Science*, vol. 3113. Springer. 2004. pp. 112–124.

[18] Jiang, T.; Ravikumar, B.: Minimal NFA Problems Are Hard. *SIAM J. Comput.*. vol. 22, no. 6. December 1993: pp. 1117–1141. ISSN 0097-5397. doi:10.1137/0222067.

[19] Lengál, O.; Šimáček, J.; Vojnar, T.: VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'12. Berlin, Heidelberg: Springer-Verlag. 2012. ISBN 978-3-642-28755-8. pp. 79–94. doi:10.1007/978-3-642-28756-5_7.

[20] Paige, R.; Tarjan, R. E.: Three Partition Refinement Algorithms. *SIAM J. Comput.*. vol. 16, no. 6. December 1987: pp. 973–989. ISSN 0097-5397. doi:10.1137/0216062.

[21] Ranzato, F.: A More Efficient Simulation Algorithm on Kripke Structures. In *MFCS*, *Lecture Notes in Computer Science*, vol. 8087. Springer. 2013. pp. 753–764.

[22] Ranzato, F.; Tapparo, F.: A New Efficient Simulation Equivalence Algorithm. In *LICS*. IEEE Computer Society. 2007. pp. 171–180.

[23] Regular Expression Library. 2017. [Online; accessed 13. 05. 2017]. Retrieved from: http://regexlib.com/

[24] Veanes, M.: Applications of Symbolic Finite Automata. In *Proceedings of the 18th International Conference on Implementation and Application of Automata*. CIAA'13. Berlin, Heidelberg: Springer-Verlag. 2013. ISBN 978-3-642-39273-3. pp. 16–23. doi:10.1007/978-3-642-39274-0_3.

[25] Veanes, M.; Bjørner, N.: Symbolic Automata: The Toolkit. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'12. Berlin, Heidelberg: Springer-Verlag. 2012. ISBN 978-3-642-28755-8. pp. 472–477. doi:10.1007/978-3-642-28756-5_33.

[26] Veanes, M.; Halleux, P. d.; Tillmann, N.: Rex: Symbolic Regular Expression Explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*. ICST '10. Washington, DC, USA: IEEE Computer Society. 2010. ISBN 978-0-7695-3990-4. pp. 498–507. doi:10.1109/ICST.2010.15.

[27] Veanes, M.; Hooimeijer, P.; Livshits, B.; et al.: Symbolic Finite State Transducers: Algorithms and Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. New York, NY, USA: ACM. 2012. ISBN 978-1-4503-1083-3. pp. 137–150. doi:10.1145/2103656.2103674.

Appendices

Appendix A

Contents of CD

Directories and files of attached CD:

- `/doc/` contains source files of this thesis,
- `/src/` contains source files of implementation of simulation algorithms,
- `/results/` contains `.csv` files of experimental results where
  - `regexNormal.csv` contains comparison of simulation algorithms for SFAs created from regular expressions,
  - `regexReverse.csv` contains comparison of simulation algorithms for reversed SFAs created from regular expressions,
  - `WS1SNormal.csv` contains comparison of simulation algorithms for SFAs created during decision procedure of WS1S logic where automata ending with `.dfa` are deterministic and thos ending with `.aut` are nondeterministic,
  - `WS1SReverse.csv` contains comparison of simulation algorithms for reversed SFAs created during decision procedure of WS1S logic,
  - `edgecase.csv` contains comparison of simulation algorithms for SFAs M_k from Example 3.2,
  - `regexSimDetMin.csv` contains comparison of deterministic minimization and reduction based on simulation of SFAs created from regular expressions,
  - `WS1SSimDetMin.csv` contains comparison of deterministic minimization and reduction based on simulation of SFAs created during decision procedure of WS1S logic.

Appendix B

Bound of the number of transitions

In this section we prove Equation 3.1:

$$\sum_{k=0}^{m} k\binom{m}{k} = m2^{m-1}.$$

Proof. To prove this equation, we need to first realize that for $n \geq 0$

$$2^n = (1+1)^n = 1^n 1^0\binom{n}{0} + 1^{n-1}1^1\binom{n}{1} + \cdots + 1^0 1^n\binom{n}{n} = \sum_{k=0}^{n}\binom{n}{k}.$$

Now

$$\begin{aligned}
\sum_{k=0}^{m} k\binom{m}{k} &= 0\binom{m}{0} + 1\binom{m}{1} + \cdots + m\binom{m}{m} = \\
&= \binom{m}{0} + \binom{m}{1} + \cdots + \binom{m}{m} - \binom{m}{0} + \\
&+ \binom{m}{0} + \binom{m}{1} + \cdots + \binom{m}{m} - \binom{m}{0} - \binom{m}{1} + \\
&+ \binom{m}{0} + \binom{m}{1} + \cdots + \binom{m}{m} - \binom{m}{0} - \binom{m}{1} - \binom{m}{2} + \\
&+ \ \vdots \ + \ \vdots \ + \ddots + \ \vdots \ - \ \vdots \ - \ \vdots \ - \ \vdots \ - \ddots + \\
&+ \binom{m}{0} + \binom{m}{1} + \cdots + \binom{m}{m} - \binom{m}{0} - \binom{m}{1} - \binom{m}{2} - \cdots - \binom{m}{m-1} = \\
&= m\sum_{k=0}^{m}\binom{m}{k} - m\binom{m}{0} - (m-1)\binom{m}{1} - \cdots - 1\binom{m}{m-1} - 0\binom{m}{m} = \\
&= m2^m - \sum_{k=0}^{m} k\binom{m}{m-k} = m2^m - \sum_{k=0}^{m} k\binom{m}{k},
\end{aligned}$$

because $\binom{m}{m-k} = \binom{m}{k}$. Let $x = \sum_{k=0}^{m} k\binom{m}{k}$. Then we have

$$\begin{aligned}
x &= m2^m - x \\
2x &= m2^m \\
x &= m2^{m-1}.
\end{aligned}$$

□