VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

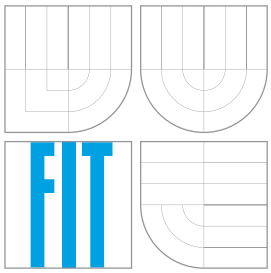
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN HRUŠKA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EFEKTIVNÍ ALGORITMY PRO PRÁCI S KONEČNÝMI AUTOMATY

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN HRUŠKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. ONDŘEJ LENGÁL
SUPERVISOR

BRNO 2013

Abstrakt

Nedeterministické konečné automaty jsou používány v mnoha oblastech informatiky, mimo jiné také ve formalní verifikaci nebo při návrhu číslicových obvodů, pro reprezentaci regulárních jazyků. Jejich výhodou oproti deterministickým konečným automatům je schopnost až exponenciálně stručnější reprezentace jazyka. Nicméně, tato výhoda může být pozbyta, jestliže je zvolen naivní přístup k implementaci některých operací, jako je například test jazykové inkluze dvojice automatů, jehož naivní implementace provádí explicitní determinizaci jednoho z automatů. V poslední době bylo ale představeno několik nových přístupů, které právě explicitní determinizaci při testu jazykové inkluze předcházejí. Tyto přístupy využívají technik tzv. antichainů nebo tzv. bisimulace vzhůru ke kongruenci. Cílém této práce je vytvoření efektivní implementace zmíněných přístupů v podobě nového rozšíření knihovny VATA. Vytvořená implementace byla otestována a je rychlejší v 90 % testovaných případů nežli jiné implementace, oproti nimž je až řádově rychlejší.

Abstract

Nondeterministic finite automata are used in many areas of computer science, including, but not limited to, formal verification or the design of digital circuits, for the representation of a regular language. Their advantages over deterministic finite automata is that they may represent a language in evan exponentially conciser way. However, this advantage may be lost if a naïve approach to some operations is taken, in particular for checking language inclusion of a pair of automata, the naïve implementation of which performs an explicit determinization of one of the automata. Recently, several new techniques for this problem that avoid explicit determinization (using the so-called antichains or bisimulation up to congruence) have been proposed. The main goal of the presented work is to efficiently implement these techniques as a new extension of the VATA library. The implementation has been evaluated and is superior to the other implementations in over 90 % of tested cases by the factor of 2 to 100.

Klíčová slova

konečné automaty, formální verifikace, jazyková inkluze, bisimulace ke kongruenci, antichain, knihovna VATA

Keywords

finite automata, formal verification, language inclusion, bisimulation up to congruence, antichains, VATA library

Citace

Martin Hruška: Efficient Algorithms for Finite Automata, bakalářská práce, Brno, FIT VUT v Brně, 2013

Efficient Algorithms for Finite Automata

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Lengála

.....
Martin Hruška
May 9, 2013

Poděkování

Rád bych tímto poděkoval vedoucímu této práce, Ing. Ondřeji Lengálovi, za odborné rady a vedení při tvorbě práce.

© Martin Hruška, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

Chapter 1

Introduction

A finite automaton (FA) is a model of computation with applications in different branches of computer science, e.g., compiler design, formal verification, the design of digital circuits or natural language processing. In formal verification alone are its uses abundant, for example in model checking of safety temporal properties [2], abstract regular model checking [5], static analysis [6], or decision procedures of some logics, such as Presburger arithmetic or weak monadic second-order theory of one successor (WS1S) [7].

Many of the mentioned applications need to perform certain expensive operations on FA, such as checking universality of an FA (i.e., checking whether it accepts any word over a given alphabet), or checking language inclusion of a pair of FA (i.e., testing whether the language of one FA is a subset of the language of the other FA). The classical (so called *textbook*) approach is based on *complementation* of the language of one of the FA. Complementation is easy for *deterministic* FA (DFA)—just swapping accepting and non-accepting states—but a hard problem for *nondeterministic* FA (NFA), which need to be determinised first (this may lead to an exponential explosion in the number of the states of the automaton). Both operations of checking of universality and language inclusion over NFA are PSPACE-complete problems [18].

Recently, there has been a considerable advance in techniques for dealing with these two problems. The new techniques are either based on the so-called *antichains* [18, 1] or the so-called *bisimulation up to congruence* [4]. In general, those techniques do not need an explicit construction of the complement automaton. They only construct a sub-automaton which is sufficient for either proving that the universality or inclusion hold, or finding a counterexample.

Unfortunately, there is currently no efficient implementation of a general NFA library that would use the state-of-the-art algorithms for the mentioned operations on automata. The closest implementation is VATA [14], a general library for nondeterministic finite *tree* automata, which can be used even for NFA (being modelled as unary tree automata) but not with the optimal performance given by its overhead that comes with the ability to handle much richer structures.

The goal of this work is two-fold: (i) extending VATA with an NFA module implementing basic operations on NFA, such as union, intersection, or checking language inclusion, and (ii) an efficient design and implementation of operations for checking language inclusion of NFA using bisimulation up to congruence (which is missing in VATA for tree automata).

After this introduction, Chapter 2 of this text describes the theoretical background. Chapter3 provides a description of recently proposed efficient approaches to language inclusion testing and their optimization. A list of the existing libraries for finite automata

manipulation is given in Chapter 4. The same chapter provides description of the VATA library. The design of the new module of the VATA library and algorithms used therein are described in Chapter 5. The implementation optimization of the algorithms for language inclusion checking and issues of other implementation is discussed in chapter 6. The evaluation of the optimized algorithms for the inclusion checking is in Chapter 7. Chapter 8 summaries the thesis and gives directions for future work.

Chapter 2

Preliminaries

This chapter contains theoretical foundations of the thesis. No proofs are given but they can be found in referenced literature [13, 9]. First, languages will be defined, then finite automata and their context and regular languages and their closure properties will follow.

2.1 Languages

We call a finite set of symbols Σ an *alphabet*. A *word* w over Σ of *length* n is a finite sequence of symbols $w = a_1 \cdots a_n$, where $\forall 1 \leq i \leq n \ . \ a_i \in \Sigma$. An *empty word* is denoted as $\epsilon \notin \Sigma$ and its length is 0. We define *concatenation* as an associative binary operation on words over Σ represented by the symbol $\cdot$ such that for two words $u = a_1 \cdots a_n$ and $v = b_1 \cdots b_m$ over Σ it holds that $\epsilon \cdot u = u \cdot \epsilon = u$ and $u \cdot v = a_1 \cdots a_n b_1 \cdots b_m$. We define Σ^* as a set of all words over Σ including the empty word and Σ^+ as set of all words over Σ without the empty word, so it holds that $\Sigma^* = \Sigma^+ \cup \epsilon$. A *language* L over Σ is a subset of Σ^*. Given a pair of languages L_1 over an alphabet Σ_1 and L_2 over an alphabet Σ_2, their concatenation is defined as $L_1 \cdot L_2 = \{x \cdot y \mid x \in L_1, y \in L_2\}$. We define *iteration* L^* and *positive* iteration L^+ of a language L over an alphabet Σ as:

- $L^0 = \{\epsilon\}$,
- $L^{n+1} = L \cdot L^n$, for $n \geq 1$,
- $L^* = \bigcup_{n \geq 0} L^n$,
- $L^+ = \bigcup_{n \geq 1} L^n$.

2.2 Finite Automata

2.2.1 Nondeterministic Finite Automaton

A nondeterministic finite automaton (NFA) is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, where

- Q is a finite set of *states*,
- Σ is an alphabet,
- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation. We use $p \xrightarrow{a} q$ to denote that $(p, a, q) \in \delta$,
- $I \subseteq Q$ is finite set of states, elements of I are called *initial states*.

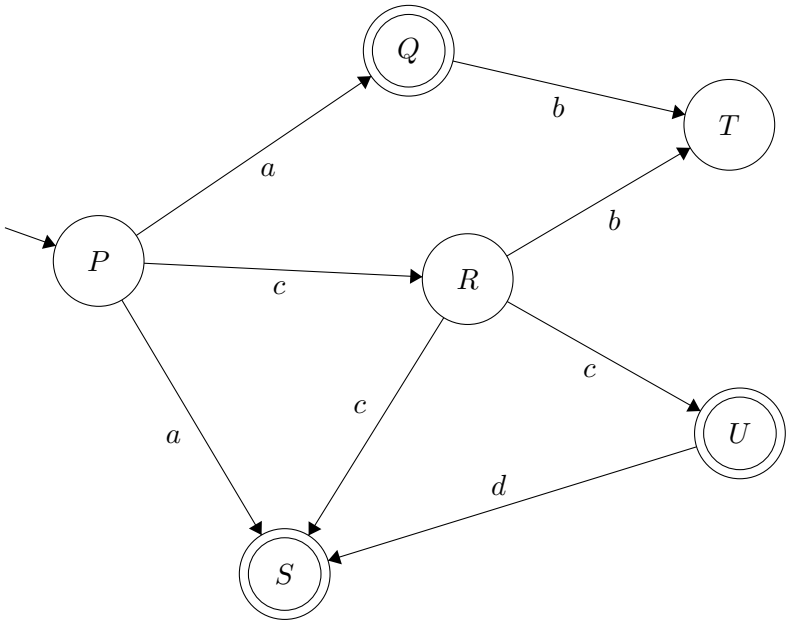


Figure 2.1: An example of an NFA

- $F \subseteq Q$ is finite set of states, elements of F are called *final states*.

An example of a NFA over $\Sigma = \{a, b, c, d\}$ is shown in the figure 2.1. Notice the nondeterministism of transitions, e.g., for state P over a.

2.2.2 Deterministic Finite Automaton

A *deterministic finite automaton* (DFA) is a special case of an NFA, where δ is a partial function $\delta : Q \times \Sigma \to Q$ and $|I| \leq 1$. To be precise, we give the whole definition of DFA.

A DFA is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- δ: $Q \times \Sigma \to Q$ is a partial transition function, we use $p \xrightarrow{a} q$ to denote that $\delta(p, a) = q$,
- $I \subseteq Q$ is finite set of initial states, such that $|I| \leq 1$,
- $F \subseteq Q$ is finite set of final states.

An example of a DFA over $\Sigma = \{a, b, c\}$ is given in the figure 2.2.

2.2.3 Operations over Finite Automata

Automata Union

Given a pair of NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$. Their union is defined by

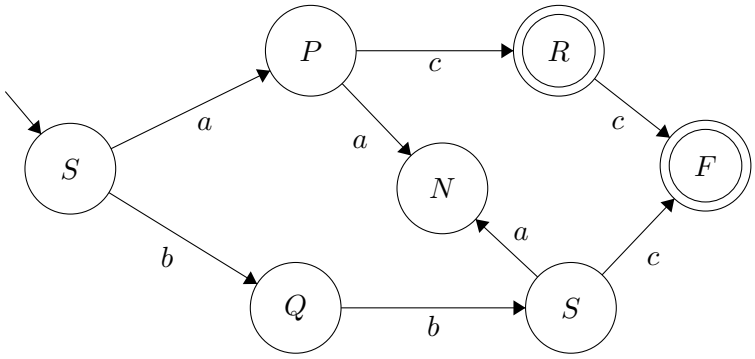


Figure 2.2: An example of an DFA

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

Note that $L_{\mathcal{A}\cup\mathcal{B}} = L_\mathcal{A} \cup L_\mathcal{B}$

Automata Intersection

Given a pair of NFA, $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$. Their intersection is defined by

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

where δ is defined by

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

Note that $L_{\mathcal{A}\cap\mathcal{B}} = L_\mathcal{A} \cap L_\mathcal{B}$

Subset construction

Now we will define how to construct equivalent DFA $\mathcal{A}_{det}$ for a given NFA $\mathcal{A} = (Q, \Sigma, \delta, S, F)$.

$\mathcal{A}_{det} = (2^Q, \Sigma, \delta_{det}, S, F_{det})$, where

- 2^Q is power set of Q
- $F_{det} = \{Q' \subseteq Q \mid Q' \cap F \neq \varnothing\}$
- $\delta_{det}(Q', a) = \bigcup\limits_{q \in Q'} \delta(q, a)$, where $a \in \Sigma$

This classical (so-called *textbook*) approach is called *subset construction*. An example of this approach is shown on the figure 2.3.

2.2.4 Run of Finite Automaton

A *run* of an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ from a state q over a word $w = a_1 \dots a_n$ is a sequence $r = q_0 \dots q_n$, where $\forall\, 0 \leq i \leq n \;.\; q_i \in Q$ such that $q_0 = q$ and $(q_i, a_{i+1}, q_{i+1}) \in \delta$. The run r is called *accepting* iff $q_n \in F$. An word $w \in \Sigma^*$ is called *accepting*, if there exists an *accepting* run for w. An *unreachable* state q of an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$

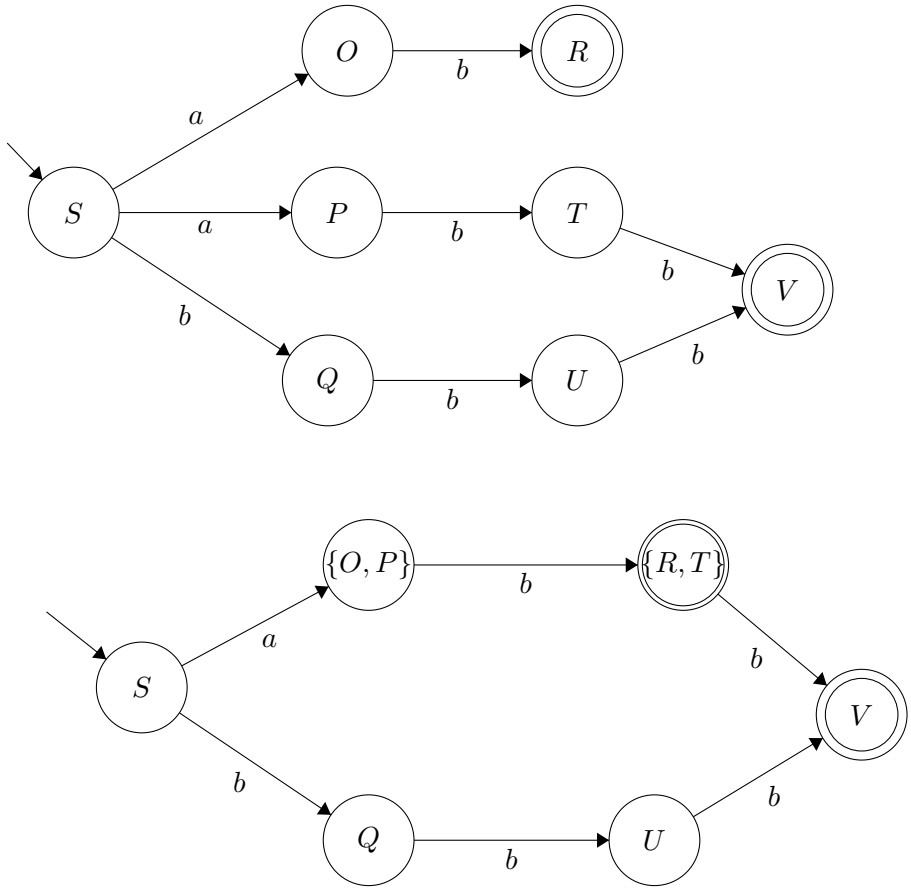


Figure 2.3: A simple example of NFA to DFA conversion via the subset construction. Here is shown small NFA with small Σ, but for larger NFA could state explosion occur.

is a state for which there is no run $r = q_0 \dots q$ of $\mathcal{A}$ over a word $w \in \Sigma^*$ such that $q_0 \in I$. An *useless* (also called nonterminating) state q of an NFA $A = (Q, \Sigma, \delta, I, F)$ is state that there is no run $r = q \dots q_n$ of A over a word $w \in \Sigma^*$ such that $q_n \in F$. Given a pair of states p, q of an NFA $A = (Q, \Sigma, \delta, I, F)$, these states are equivalent if $\forall w \in \Sigma^* :$ *Run from* p *over* w *is accepting* $\Leftrightarrow$ *Run from* q *over* w *is accepting.*

2.2.5 Complete DFA

Complete DFA $\mathcal{A}_\mathcal{C} = (Q_C, \Sigma, \delta_C, I_C, F_C)$ is DFA where $\forall p \in Q_C \, \forall a \in \Sigma \, \exists q \in Q_C : p \xrightarrow{a} q \in \delta_C$. It is possible to create complete DFA $\mathcal{A}_\mathcal{C} = (Q_C, \Sigma, \delta_C, I_C, F_C)$ from DFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ such that $Q_C = Q \cup \{q\}$, $I_C = I$, $F_C = F$, $\delta_C = \delta \cup \{p \xrightarrow{a} q \,|\, a \in \Sigma, p \in Q_C, p \xrightarrow{a} r \notin \delta, r \in Q\}$.

2.2.6 Minimal DFA

Minimal DFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is complete DFA which satisfies this conditions:

- There are no unreachable states
- There is maximal one nonterminating state

- Equivalent states are collapsed

2.2.7 Language of Finite Automaton

Let have a NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$. The *language* of a state $q \in Q$ is defined as $L_{\mathcal{A}}(q) = \{w \in \Sigma^* \mid \textit{there exists an accepting run of } \mathcal{A} \textit{ from } q \textit{ over } w\}$, while the language of a set of states $R \subseteq Q$ is defined as $L_{\mathcal{A}}(R) = \bigcup_{q \in R} L_{\mathcal{A}}(q)$. The language of an NFA $\mathcal{A}$ is defined as $L_{\mathcal{A}} = L_{\mathcal{A}}(I)$.

2.3 Regular Languages

A language L is *regular*, if there exists an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, such that $L = L_{\mathcal{A}}$.

2.3.1 Closure Properties

Regular languages are closed under certain operation if result of this operation over some regular language is always regular language too.

Let introduce the closure properties of regular languages on an alphabet Σ:

- Union: $L_1 \cup L_2$
  Union of two NFA is described in section 2.2.3.

- Intersection: $L_1 \cap L_2$
  Intersection of two NFA is described in section 2.2.3.

- Complement: $\overline{L}$
  Complement of NFA $\mathcal{A}$ is done by its determinizing (via subset construction described in section 2.2.3), completion of this DFA (via method described in 2.2.5) and switching its final and non-final states set.

- Difference: $L_1 - L_2$
  Difference of NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ and NFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$ is done by creating complete DFA $\mathcal{A}_C$ and $\mathcal{B}_C$ (by methods 2.2.3 and 2.2.5) and creating product DFA (created as is described in section **??**) $\mathcal{A}_C \times \mathcal{B}_C = (Q_{\mathcal{A}_C} \times Q_{\mathcal{B}_C}, \Sigma, \delta_{\mathcal{A}_C \times \mathcal{B}_C}, I_{\mathcal{A}_C} \times I_{\mathcal{B}_C}, F)$ where $F = \{(p, q) \mid p \in F_{\mathcal{A}_C} \wedge q \notin F_{\mathcal{B}_C}\}$.

- Reversal: $\{a_1 \dots a_n \in L \mid y = a_n \dots a_1 \in L\}$
  Reversion of a NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ is NFA $\mathcal{A}_{rev} = (Q_{\mathcal{A}_{rev}}, \Sigma, \delta_{\mathcal{A}_{rev}}, I_{\mathcal{A}_{rev}}, F_{\mathcal{A}_{rev}})$ where $Q_{\mathcal{A}_{rev}} = Q_{\mathcal{A}}$, $I_{\mathcal{A}_{rev}} = F_{\mathcal{A}}$, $F_{\mathcal{A}_{rev}} = I_{\mathcal{A}}$ and $\delta_{\mathcal{A}_{rev}} = \{(q, a, p) \mid p, q \in Q_{\mathcal{A}}, a \in \Sigma, (p, a, q) \in \delta_{\mathcal{A}}\}$.

- Iteration: L^*
  Iteration of a NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ is NFA $\mathcal{A}^* = (Q_{\mathcal{A}}, \Sigma, \delta^*_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ where $\delta^*_{\mathcal{A}} = \delta_{\mathcal{A}} \cup \{(i_{\mathcal{A}}, \epsilon, f_{\mathcal{A}}) \mid i_{\mathcal{A}} \in I_{\mathcal{A}}, f_{\mathcal{A}} \in F_{\mathcal{A}}\} \cup F_{\mathcal{A}} \times \{\epsilon\} \times I_{\mathcal{A}}$

- Concatenation: $L \cdot K = \{x \cdot y \mid x \in L \wedge y \in K\}$
  Concatenation of a NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ and a NFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$ is NFA $\mathcal{A} \cdot \mathcal{B} = (Q_{\mathcal{A}} \cup Q_{\mathcal{B}}, \Sigma, \delta, I_{\mathcal{A}}, F_{\mathcal{B}})$ where $\delta = \delta_{\mathcal{A}} \cup \delta_{\mathcal{B}} \cup F_{\mathcal{A}} \times \{\epsilon\} \times I_{\mathcal{B}}$.

Chapter 3

Inclusion Checking over NFA

Given a pair of NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$, the *language inclusion problem* is decision whether $L_\mathcal{A} \subseteq L_\mathcal{B}$ what is defined by standard set operations as $L_\mathcal{A} \cap \overline{L_\mathcal{B}} = \varnothing$. This problem is PSPACE-complete [18]. The *textbook* algorithm for checking inclusion $L_\mathcal{A} \subseteq L_\mathcal{B}$ works by first determinizing $\mathcal{B}$ (yielding the DFA $\mathcal{B}_{det}$ using subset construction algorithm 2.2.3), complementing it ($\overline{\mathcal{B}_{det}}$) and constructing the NFA $\mathcal{A} \times \overline{\mathcal{B}_{det}}$ accepting the intersection of $L_\mathcal{A}$ and $L_{\overline{\mathcal{B}_{det}}}$ and checking whether its language is nonempty. Any accepting run in this automaton may serve as a witness that the inclusion between $\mathcal{A}$ and $\mathcal{B}$ does not hold. Some recently introduced approaches (so-called antichains [18], its optimization using simulation [1] and so-called bisimulation up to congruence [4]) avoid the explicit construction of $\overline{\mathcal{B}_{det}}$ and the related state explosion in many cases.

We have to define following terms for the further description of the new techniques for the inclusion checking. We denote product state of an NFA $\mathcal{A} \times \mathcal{B}$ as a pair (p, P) of a state $p \in Q_\mathcal{A}$ and a macrostate $P \subseteq Q_\mathcal{B}$. We define post-image of the product state (p, P) of a NFA $\mathcal{A} \times \mathcal{B}$ by: $Post((p, P)) := \{(p', P') \mid \exists a \in \Sigma : (p, a, p') \in \delta_\mathcal{A}, P' = \{p'' \mid \exists p \in P : (p, a, p'') \in \delta_\mathcal{B}\}\}$

3.1 Checking Inclusion with Antichains and Simulation

We define an antichain, simulation and some others terms before describing the algorithm itself.

Given a partially ordered set Y, an *antichain* is a set $X \subseteq Y$ such that all elements of X are incomparable.

A forward *simulation* on the NFA $\mathcal{A}$ is a relation $\preceq \subseteq Q_1 \times Q_1$ such that if $p \preceq r$ then (i) $p \in F_1 \Rightarrow r \in F_1$ and (ii) for every transition $p \xrightarrow{a} p'$, there exists a transition $r \xrightarrow{a} r'$ such that $p' \preceq r'$. Note that simulation implies language inclusion, i.e., $p \preceq q \Rightarrow L_\mathcal{A}(p) \subseteq L_\mathcal{A}(q)$ [8].

For two macro-states P and R of a NFA is $R \preceq^{\forall\exists} P$ shorthand for $\forall r \in R. \exists p \in P : r \preceq p$.

Product state (p, P) is accepting, if p is accepting in automaton A and P is rejecting in automaton B.

3.1.1 Antichain Algorithm Description

The antichains algorithm [18] starts searching for a final state of the automaton $\mathcal{A} \times \overline{\mathcal{B}_{det}}$ while pruning out the states which are not necessary to explore. $\mathcal{A}$ is explored nondeterministically and $\mathcal{B}$ is gradually determinized, so the algorithm explores pairs (p, P) where

$p \in Q_{\mathcal{A}}$ and $P \subseteq Q_{\mathcal{B}}$. The antichains algorithm derives new states along the product automaton transitions and inserts them to the set of visited pairs X. X keeps only minimal elements with respect to the ordering given by $(r, R) \sqsubseteq (p, P)$ iff $r = p \wedge R \subseteq P$. If there is generated a pair (p, P) and there is $(r, R) \in X$ such that $(r, R) \sqsubseteq (p, P)$, we can skip (p, P) and not insert it to X for further search.

An improvement of the antichains algorithm using simulation [1] is based on the following optimization. We can stop the search from a pair (p, P) if either (a) there exists some already visited pair $(r, R) \in X$ such that $p \preceq r \wedge R \preceq^{\forall\exists} P$, or (b) there is $p' \in P$ such that $p \preceq p'$. This first optimization is in algorithm 1 at lines 11–14.

Another optimization [1] of the antichain algorithm is based on the fact that $L_{\mathcal{A}}(P) = L_{\mathcal{A}}(P - \{p_1\})$ if there exists $p_2 \in P$, such as $p_1 \preceq p_2$. We can remove the state p_1 from macrostate P, because if $L_{\mathcal{A}}(P)$ rejects the word then $L_{\mathcal{A}}(P - \{p_1\})$ rejects this word too. This optimization is applied by the function $Minimize$ at the lines 4 and 7 in the algorithm 1.

The whole pseudocode of the antichain algorithm is given as algorithm 1.

Algorithm 1: Language inclusion checking with antichains and simulations

Input: NFA's $\mathcal{A} = (Q_A, \Sigma, \delta_A, I_A, F_A)$, $\mathcal{B} = (Q_B, \Sigma, \delta_B, I_B, F_B)$.
A relation $\preceq \in (\mathcal{A} \cup \mathcal{B})^{\subseteq}$.
Output: TRUE if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Otherwise, FALSE.

```
1  if there is an accepting product-state in {(i, I_B) | i ∈ I_A} then
2      return FALSE;
3  Processed:=∅;
4  Next:= Initialize({(s, Minimize(I_B)) | s ∈ I_A});
5  while (Next ≠ ∅) do
6      Pick and remove a product-state (r, R) from Next and move it to Processed;
7      forall the (p, P) ∈ {(r', Minimize(R')) | (r', R') ∈ Post((r, R))} do
8          if (p, P) is an accepting product-state then
9              return FALSE;
10         else
11             if ∄p' ∈ P s.t. p ⪯ p' then
12                 if ∄(x, X) ∈ Processed ∪ Next s.t. p ⪯ x ∧ X ⪯^{∀∃} P then
13                     Remove all (x, X) from Processed ∪ Next s.t. x ⪯ p ∧ P ⪯^{∀∃} X;
14                     Add (p, P) to Next;
15 return TRUE;
```

3.2 Checking Inclusion with Bisimulation up to Congruence

Another approach to checking language inclusion of NFA is based on bisimulation up to congruence [4]. The definition of congruence relation is following:

Let X be a set with a n-ary operation O over X. Congruence is an equivalence relation R, which follows this condition $\forall a_1, \ldots, a_n, b_1, \ldots, b_n \in X$:

$$a_1 \sim_R b_1, \ldots, a_n \sim_R b_n \Rightarrow O_n(a_1, \ldots, a_n) \sim_R O_n(b_1, \ldots, b_n), \text{ where } a_i \in X, b_i \in X$$

This technique was originally developed for checking equivalence of languages of automata but it can also be used for checking language inclusion, based on the observation

that $L_{\mathcal{A}} \cup L_{\mathcal{B}} = L_{\mathcal{B}} \Leftrightarrow L_{\mathcal{A}} \subseteq L_{\mathcal{B}}$.

This approach is based on the computation of a *congruence closure* $c(R)$ for some binary relation on states of the determinized automaton $R \subseteq 2^Q \times 2^Q$ defined as a relation $c(R) = (r \cup s \cup t \cup u \cup id)^\omega(R)$, where

$$id(R) = R,$$

$$r(R) = \{(X, X) \mid X \subseteq Q\},$$

$$s(R) = \{(Y, X) \mid XRY\},$$

$$t(R) = \{(X, Z) \mid \exists Y \subseteq Q,\ XRYRZ\},$$

$$u(R) = \{(X_1 \cup X_2, Y_1 \cup Y_2) \mid X_1RY_1 \wedge X_2RY_2\}.$$

3.2.1 Congruence Algorithm Description

The congruence algorithm works on a similar principle as the antichains algorithm but it starts building not only $\mathcal{B}_{det}$ but also $\mathcal{A}_{det}$ because the original purpose of this algorithm is checking of language equivalence. States of a product automaton $\mathcal{A}_{det} \times \mathcal{B}_{det}$ (so-called product states) are the pairs $(P_{\mathcal{A}}, P_{\mathcal{B}})$ of macrostate $P_{\mathcal{A}} \subseteq Q_{\mathcal{A}}$ and macrostate $P_{\mathcal{B}} \subseteq Q_{\mathcal{B}}$. The algorithm searches for a victim that proves $L_{\mathcal{A}} \neq L_{\mathcal{B}}$. The victim is a product state $(P_{\mathcal{A}}, P_{\mathcal{B}})$ which breaks a condition that the $P_{\mathcal{A}}$ contains a final state of A if and only if $P_{\mathcal{B}}$ contains a final state of B.

The optimization brought by this algorithm is based on computing of a congruence closure of the set of already visited pairs of macrostates. If the generated pair is in the congruence closure, it can be skipped and further not processed. The whole pseudocode of the congruence algorithm is given as algorithm 2.

Algorithm 2: Language equivalence checking with congruence

Input: NFA's $A = (Q_A, \Sigma, \delta_A, I_A, F_A),\ B = (Q_B, \Sigma, \delta_B, I_B, F_B)$.
Output: TRUE, if $L(A)$ and $L(B)$ are in equivalence relation. Otherwise, FALSE.

```
1  Processed := ∅;
2  Next := (I_A, I_B);
3  while Next ≠ ∅ do
4      Pick and remove a product state (X, Y) from Next;
5      if (X,Y) ∈ c(Processed ∪ Next) then
6          skip;
7      if ¬({x ∈ X|x ∈ F_A} ≠ ∅ ⇔ {y ∈ Y|y ∈ F_B} ≠ ∅) then
8          return FALSE;
9      Add (post(X, Y)) to Next;
10     Add(X,Y) to Processed;
11 return TRUE;
```

Comparing the mentioned approaches to the checking language inclusion can be seen in Figure 3.1.

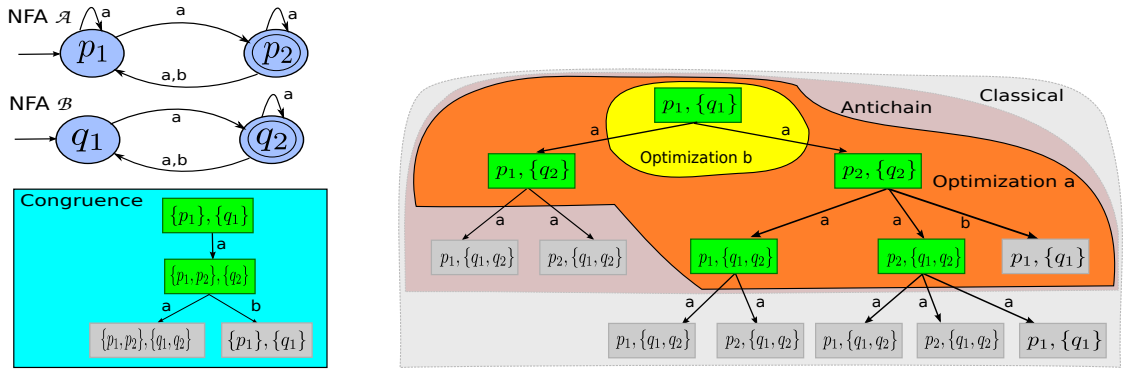


Figure 3.1: The figure is based on an example from [1]. It shows the procedure of checking language inclusion between two NFA using the mentioned approaches (which correspond to the labeled areas). The antichain algorithm reduces number of the generated states compared with the classical, e.g., $(p_2, \{q_1, q_2\})$ is not further explored because $(p_2, \{q_2\}) \sqsubseteq (p_2, \{q_1, q_2\})$. The optimization a and b are improvements of the antichain algorithm using simulation. The congruence algorithm also reduces number of the generated states, so $(\{p_1, p_2\}, \{q_1, q_2\})$ is not further explored because it is in congruence closure of the set of visited states.

3.2.2 Computation of Congruence Closure

The computation of the congruence closure is crucial for performance and efficiency of the whole method. This thesis implements an algorithm described by [4] which is based on using of the so-called rewriting rules. For each pair of macrostates (X, Y) in a relation R of the visited macrostates exists two rewriting rules which has following form:

$$X \to X \cup Y \qquad\qquad Y \to X \cup Y$$

These rules can be used for computation of a *normal form* of a set of states [4]. The normal form of a macrostate X created with the usage of rewriting rules of the relation R is denoted as $X{\downarrow_R}$. Checking if $(X, Y) \in c(R)$ using derivation of the normal form is based on the observation that $X{\downarrow_R} = Y{\downarrow_R}$ iff $(X, Y) \in c(R)$ [4].

An example (taken from [4]) is given to illustrate an application of this approach for checking equivalency of NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$ (both NFA are on figure 3.2). Let have a relation $R = \{(\{x\}, \{u\}), (\{y, z\}, \{u\})\}$ of the visited product states and a newly generated product state $(\{x, y\}, \{u\})$ (where $\{x, y\} \subseteq Q_\mathcal{A}$ and $\{u\} \subseteq Q_\mathcal{B}$). For checking of $(\{x, y\}, \{u\}) \in c(R)$ it is needed to compute the normal forms of the macrostates $\{x, y\}$ and $\{u\}$ A derivation of both normal forms is shown on 3.3. The normal form of the set $\{x, y\}$ is derived in two steps. At the first step is applied rule $\{x\} \to \{x, u\}$ (base on $(\{x\}, \{u\}) \in R$) so we get a set $\{x, y, u\}$. As the second one, rule $\{u\} \to \{y, z, u\}$ (based on product state $(\{y, z\}, \{u\}) \in R$) is applied, so the result is $\{x, y, z, u\}$. The normal form of the set $\{u\}$ is derived in two steps too. At the first step, a rule $\{u\} \to \{x, u\}$ is applied so we get a set $\{x, u\}$ and then rule $\{u\} \to \{y, z, u\}$ is used and the result set is $\{x, y, z, u\}$. The derived normal sets are equal so it holds that $(\{x, y\}, u) \in c(R)$ and it is not necessary to further explore product automaton $\mathcal{A} \times \mathcal{B}$ from this state.

Problem of this approach is that we do not know which rules of relation R to use, in which order to use the and each rule can be used only once for computing a normal form.

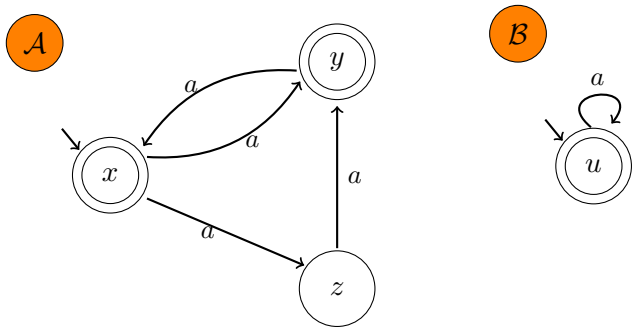


Figure 3.2: The figure shows two NFA $\mathcal{A}$, $\mathcal{B}$ which are used in example describing computation of a congruence closure in figure 3.3

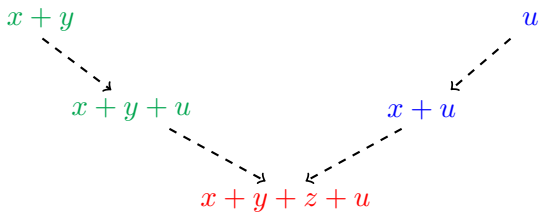


Figure 3.3: The figure (taken from [4]) shows the deriving of the normal forms of the sets $\{x, y\}$ and $\{u\}$ using rewriting rules of the macrostates of a relation $R = \{(\{x\}, \{u\}), (\{y, z\}, \{u\})\}$.

Due this conditions the time complexity for finding one rule is in the worst case rn, where $r = |R|$ and $n = |Q|$ where Q is a set of states of a NFA. The whole derivation of the normal set is bounded by complexity r^2n because we apply maximally r rules [4].

Optimization for Inclusion Checking

Since the algorithm based on bisimulation up to congruence is primarily used for checking equivalence of NFA it is possible to make some simplifications for checking inclusion. An optimization is possible in checking whether macrostate (X, Y) is in congruence closure of a relation R of the visited product states. The optimization is based on the fact that when one checks inclusion between NFA $\mathcal{A}$ and $\mathcal{B}$ it is done by checking if $\mathcal{A} \cup \mathcal{B} = \mathcal{B}$ so in all product states (X, Y) is X set of states of NFA $\mathcal{A} \cup \mathcal{B}$ and Y set of states of NFA $\mathcal{B}$. Since the states of $\mathcal{B}$ are already in macrostate X it is useful to use the rewriting rules only in following form [4]:

$$Y \rightarrow X \cup Y$$

During checking inclusion of two NFA is not also necessary to achieve $X{\downarrow_R} = Y{\downarrow_R}$ to prove that $(X, Y) \in c(R)$ but just $X \subseteq Y{\downarrow_R}$ to prove that $(X \cup Y, Y) \in c(R)$ [4].

As an example is given computation of congruence closure during checking inclusion between NFA $\mathcal{A}$ and $\mathcal{B}$ (both are on the figure 3.2). Let have a relation of visited product states $R = \{(\{x, u\}, \{u\}), (\{y, z, u\}, \{u\})\}$ and newly generated product state $(\{x, y, u\}, \{u\})$. The derivation of the normal form of the set $\{u\}$ is shown on the figure 3.4. The normal form of the set $\{u\}$ is derived in two steps, first is applied rule $\{u\} \rightarrow \{x, u\}$ (based on

Figure 3.4: The figure shows the deriving of the normal form the set $\{u\}$ using rewriting rules of the elements of a relation $R = \{(\{x, u\}, \{u\}), (\{y, z, u\}, \{u\})\}$.

$(\{x, u\}, \{u\}) \in R$) so we get a set $\{x, u\}$. Then rule $\{u\} \rightarrow \{y, z, u\}$ (based on $(\{y, z, u\}, u)$) is used and the finally derived normal form is set $(\{x, y, z, u\})$ and because the set $\{x, y, u\}$ is subset of the derived set it holds that $(\{x, y, u\}, \{u\}) \in c(R)$.

Chapter 4

Existing Finite Automata Libraries and the VATA Library

There are many different libraries for finite automata. These libraries have been created for various purposes and are implemented in different languages. At this chapter, some libraries will be described. Described libraries are just examples which represents typical disadvantages of existing libraries like classical approach for language inclusion testing which needs the determinisation of finite automaton.

At the second part of this chapter, VATA library for *tree* automata will be introduced. Library design will be briefly described and also the operations for tree automata and the plans for an extension of VATA library.

4.1 Existing Finite Automata Libraries

4.1.1 dk.brics.automaton

dk.brics.automaton is an established Java package available under the BSD license. The latest version of this library (1.11-8) was released on September 7th, 2011. Library can be downloaded and more information are on its webpage [16].

Library can use as input regular expression created by the Java *RegEx* class. It supports manipulation with NFA and DFA. Basic operation like union, intersection, complementation or run of automaton on the given word etc., are available.

Test of language inclusion is also supported but if the input automaton is NFA, it needs to be converted to DFA. This is made by *subset construction* approach which causes state explosion.

dk.brics.automaton was ported to another two languages in two different libraries, which will be described next.

libfa

libfa is a C library being part of *Augeas* tool. Library is licensed under the LGPL, version 2 or later. It also support both versions of finite automata, NFA and DFA. Regular expressions could serve like input again. libfa can be found and downloaded on it webpage [15]. libfa has no explicit operation for inclusion checking, but has the operations for intersection and complement of automata which can serve for the inclusion checking. Main disadvantage of libfa is again the need of the explicit determinisation during inclusion checking.

Fare

Fare is a library, which brings dk.brics.automaton from Java to .NET. This library has the same characteristics as dk.brics.automaton or libfa and disadvantage in need of determinisation is still here. Fare can be found on its webpage [3].

4.1.2 The RWHT FSA toolkit

The *RWHT FSA* is a toolkit for manipulating finite automata described in [10]. The latest version is 0.9.4 from year 2005. The toolkit is written in C++ and available under its special license, derived from Q Public License v1.0 and the Qt Non-Commercial License v1.0. Library can be downloaded from [11].

The RWHT FSA does not support only the classical finite automata, but also automata with weighted transitions so the toolkit has wider range of application. The toolkit implements some techniques for better computation efficiency. E.g., it supports on-demand computation technique for operations over finite automata so not all computations are evaluated immediately but some are not computed until their results are really needed. Usage of this technique leads to better memory efficiency.

The RWHT FSA toolkit does not support language inclusion checking explicitly, but contains operations for intersection, complement and determinisation which can be exploited for testing inclusion. This brings again the disadvantage of a state explosion during the explicit determinization.

4.1.3 Implementation of the State-of-the-art Algorithms

There have been recently introduced some new efficient algorithms for inclusion checking which are dealing with problem of a state explosion because they avoid the explicit determinization of a finite automaton. These algorithms have been described in section 3. All of the mentioned state-of-the-art algorithms were implemented in OCaml language for testing and evaluation purposes.

The algorithms using the antichains are possible to use not only for finite automata but also for tree automata([18, 1]). The algorithms for tree automata are provided by the VATA library which is implemented in C++ what brings the greater efficiency compared to OCaml implementation. A description of this library will be placed in next section. Despite the fact that a C++ implementation could be more efficient then OCaml implementation, there is currently no library or toolkit similar to VATA library providing efficient implementation of these algorithms for language inclusion checking over NFA.

4.2 VATA library

VATA is a highly efficient open source library for *nondeterministic tree* automata licensed under GPL, version 3. Main application of VATA is formal verification [14]. VATA library is implemented in C++ and uses the Boost C++ library. The library can be downloaded from its website [1].

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

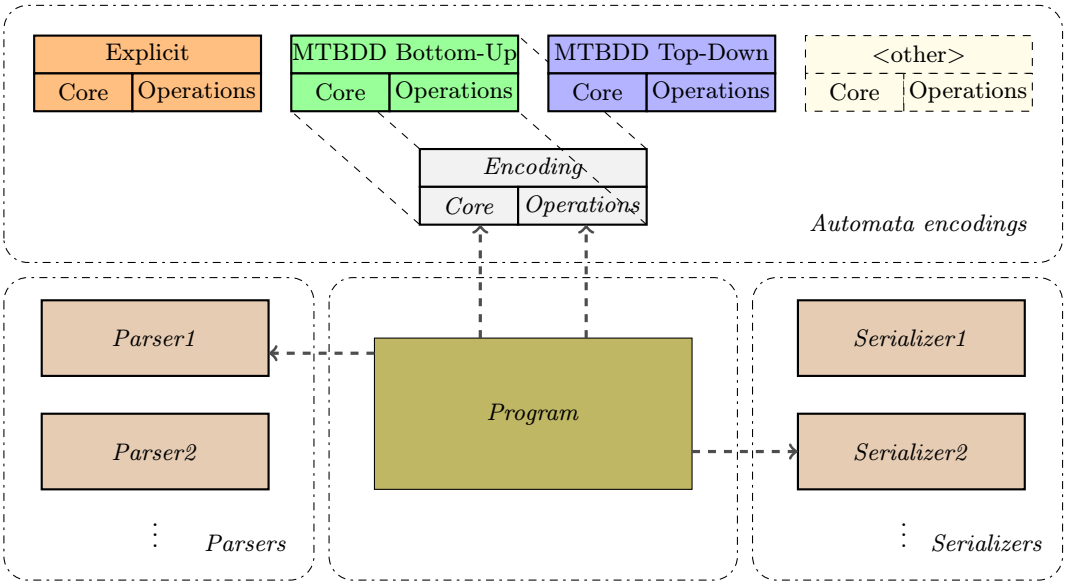


Figure 4.1: The VATA library design. The image is taken from [14]

.

4.2.1 Design

VATA provides two kind of encoding for tree automata – Explicit Encoding (top-down) and Semi-symbolic encoding (top-down and bottom-up). The main difference between encoding is in data structure for storing transition of tree automata. Semi-symbolic encoding is primary for automata with large alphabets.

The main idea of the design of VATA library is show on the image 4.1 and here is also brief description of it. An input automaton is processed by one of the parsers (currently is implemented only Timbuk format parser). A result of parsing is a data structure with the general information about automaton (the data structure stores a list of transitions of a given automaton, its final states etc.). The main program choose one of the internal encodings of the automaton. The encodings differs by a data structure they use for a representation of the automaton. Each encoding also provides the functions for transformation of the automaton from the data structure given by parser to the data structure used by chosen encoding. The encodings also implements the operations over automata. When the automaton is processed it is often dumped to output format. This is done by one of the serializers (currently there is implemented only the Timbuk format serializer too) which takes as input the same data structure which uses parser.

As you can see on the figure 4.1, the VATA library is written in a modular way, so it is easy to make an extension for finite automata. Thanks to the modularity, any new encoding can share other parts of library such as parser or serializer [14]. The VATA library also provides a command line interface which is shared by different encodings.

Explicit Encoding

The explicit encoding supports storing the transitions in top-down direction (transitions are in form $q \xrightarrow{a} (q_1, ..., q_n)$). The transitions are stored in a *hierarchical data structure based on hash tables*. First level of the data structure is hash table that maps the states

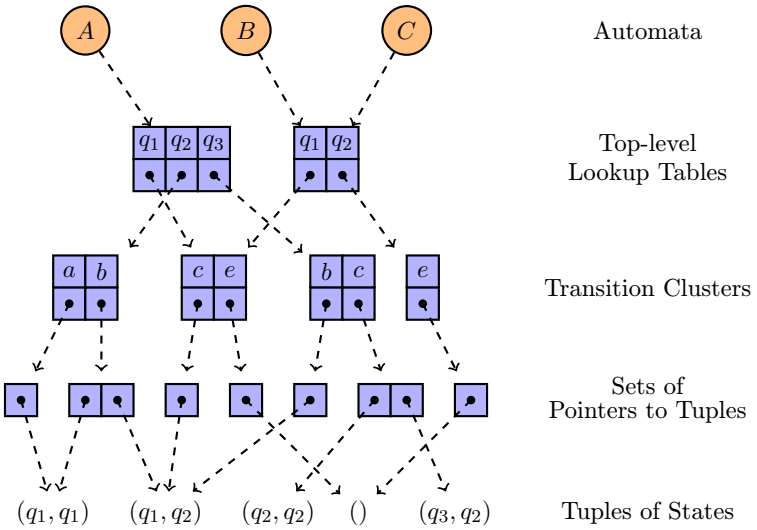


Figure 4.2: The data structure for storing transitions of the tree automaton. There is a hash table (top-level look-up table) which map a state to the pointer to another hash table (transition cluster). Transition cluster maps a symbols of input alphabet to the pointer to the set of pointers to the tuples of states.

to *transition cluster*. This clusters are also look-up tables and maps symbols of an input alphabet to a set of pointers (stored as *red-black tree*) to tuples of states. Storing tuples of states can be very memory demanding, so each tuple is stored only once and is pointed by different transitions. Inserting new transition to this structure requires a constant number of steps (exception is the worst case scenario) [14]. This data structure can be seen on figure 4.2.

For better performance is used *copy-on-write* technique [14]. The principle of this technique is that on copy of automaton is created just new pointer to transition table of original automaton and after adding a new state to one of the automaton is modified only part of the whole shared transition table.

Semi-symbolic Encoding

Transition functions in semi-symbolic encoding are stored in *multi-terminal binary decision diagrams* (MTBDD), which are extension of *binary decision diagrams*. There are provided top-down (transitions are in form $q \xrightarrow{a} (q_1, ..., q_n)$, for a with arity n) and bottom-up (transitions are in form $(q_1, ..., q_n) \xrightarrow{a} q$) representation of tree automata in semi-symbolic encoding. The specific part is the saving of symbols in MTBDD. In top-down encoding, the input symbols are stored in MTBDD with their arity, because we need to be able to distinguish between two instances of same symbols with different arity. In opposite case, bottom-up encoding does not need to store arity, because it is possible to get it from arity of tuple on left side of transition [14].

For purposes of VATA library was implemented new MTBDD package which improved the performance of library.

Operations

There are supported basic operations over tree automata like union, intersection, elimination of unreachable states, but also some advance a algorithms for inclusion checking, computation of simulation relation, language preserving size reduction based on simulation equivalence.

Optimized algorithms for inclusion testing [18, 1] are implemented. The inclusion operation is implemented in more versions, so it is possible to use only some heuristic and compare different results.

The efficiency of advanced operations does not come only from the usage of the state-of-the-art algorithms, but there are also some implementation optimization like *copy-on-write* principle for automata copying (briefly described in section 4.2.1), buffering once computed clusters of transitions, etc. Other optimization could be found in exploitation of polymorphism using C++ function templates, instead of virtual method because a call of virtual function leads to indirect function call using look-up in virtual-method table (because compiler does not know, which function will be called in runtime) what brings an overhead compared to classical direct function call and it also precludes compiler's optimizer to perform some operations [14].

More details about implementation optimization can be found in [14].

Especially advanced operations are able only for specific encoding. Some of operations implemented in VATA library and their supported encodings are in the table 4.1.

	Explicit	**Semi-symbolic**	
Operation	**top-down**	**bottom-up**	**top-down**
Union	+	+	+
Intersection	+	+	+
Complement	+	+	+
Removing useless states	+	+	+
Removing unreachable states	+	+	+
Downward and Upward Simulation	+	−	+
Bottom-Up Inclusion	+	+	−
Simulation over LTS[2]	+	−	−

Table 4.1: Table shows which operations are supported for the tree automata in the encodings implemented in VATA library.

4.2.2 Extension for Finite Automata

The purposes of the VATA library are similar as purposes of this work and because the VATA library is written in modular way, it is easy to extend it by another module, so it was decided not to create a brand new library but implement a new extension of VATA for finite automata.

The main goal is to provide efficient implementation of the operation for checking of the language inclusion using state-of-the-art algorithms. To be precise, VATA library could be already used for finite automata which can be represented by one dimensional tree automata. But the VATA library data structures for manipulating tree automata are

[2]LTS – Labeled Transitions System

designated for more complex data structures and new special implementation for finite automata will be definitely more efficient. Not only the inclusion checking algorithms will be implemented but also the algorithms for basic operations like union, intersection, removing unreachable or useless states. The new extension will implement only the explicit encoding of finite automata. The extension will use some already implemented features of the VATA library like parser and serializer or computation of simulation over states of an automaton.

Chapter 5

Design

This chapter is primarily about design of the newly created extension of VATA library for finite automata. At first, the data structures used for storing a finite automaton will be described then a principle of translation of the states and the symbols of a NFA to internal representation. The choice of an input format and its modification are justified. The algorithms for basic operations over NFA like union, intersection or removing unreachable states, etc., are given at the end of the chapter.

5.1 Data Structures for Explicit Encoding of Finite Automata

The encodings for tree automata used in VATA library differ mainly in a data structure used for storing transitions of tree automata. The explicit encoding for finite automata is defined by a data structure used for storing the transitions too. This data structure is also crucial performance of the algorithms so it is important to good analyze and design it.

5.1.1 Analysis

A NFA is defined by set of its states, its start and final states (which are subset of all states of the NFA) and also its transitions and the input alphabet (a formal definition is given in section 2.2.1). One needs to keep information about sets of start and final states to be able to distinguish between them and the other states. But it is not necessary to store the whole set of states alone because states are used only within transitions. This also holds for an input alphabet of the NFA.

The transitions keep the most information about a NFA and are also often used during the operations over NFA, so the the performance of these operations strongly depend on the efficiency of data structure for a set of transitions. For example, in many operations over NFA one wants to get all transitions for a given state and a given alphabet symbol and it is important for the efficiency of the algorithms to get those transitions in as few steps as possible. The similar needs are in the case of tree automata when it is not necessary to hold the whole set of the states but it is important to have an efficient data structure for representing transitions of the tree automata.

The data structure used for storing transitions of a tree automaton in the VATA library was described earlier in section 4.2.1 and can be seen on the figure 4.2. The evaluation of the VATA library [14] proves the efficiency of this data structure so it was decided to modify it and implement the modification in the extension of the VATA library for finite automata.

5.1.2 Design of Data Structure for Transitions of NFA

A data structure for storing transitions of a NFA is based on hash tables. The first hash table (top-level hash table) maps a given state to the pointer to a transition cluster. The transition cluster is another hash table which maps a given symbol of the input alphabet to a set of states accessible from the given state under the given symbol. Described data structure is on the figure 5.1.

The data structure for storing of the finite automata transitions is simplification of the data structure for the tree automata. Since a tree automaton's transition has following form: $q \xrightarrow{a} (q_1, \ldots, q_n)$ where $q, q_1 \ldots q_n$ are states of the tree automaton and a is the symbol of the input alphabet of the tree automaton and a finite automaton has transition in form: $q_1 \xrightarrow{a} q_2$ where q_1, q_2 are states of the finite automaton and a is its symbol, the simplification of data structure is possible because the tree version has to store the whole tuples. These tuples can be very large and it is more efficient to store them only once in a cache and in the data structure for transitions work only with pointer to a tuple instead of the tuple alone.

In case of finite automata this advantage disappears because there are no tuples of state but only states alone and keeping pointer to one state will not bring any memory efficiency (a size of a pointer to a state and the state represented by integer alone is quite similar). This causes that for the data structure for finite automata is not needed to use anything such as a set of pointers to tuples, but could be directly used a set of states instead of a set of pointers. The set of states would be pointed from transition cluster and would contain all states accessible from a given state under a certain symbol of the input alphabet.

But there is possible another simplification. The set of states does not need to be in a special set pointed by transition cluster but can be integrated to the transition cluster. When this optimization is applied, transition cluster maps symbol directly to the set of states accessible under this symbol.

The mentioned optimization enables simplification from the four levels of data structure for tree automata to the two levels data structure used for finite automata what brings simpler and more efficient manipulation with these data structure. A comparison of data structure for the finite automata and the tree automata can be seen on the figures 5.1 and figure 4.2.

This data structure apply also the copy-on-write principle for better memory efficiency so the look-up tables and the transition clusters are shared among NFA when they are same and a new look-up table and a transition is created only when a new item is inserted to the one of the automata. For example, NFA $\mathcal{A}$ and NFA $\mathcal{B}$ from the figure 5.1 are sharing the same data structure.

Let give the examples for searching and inserting a transition to this data structure for the NFA on the figure 5.1. If one wants to find all accessible states for state q_1 and symbol a in a NFA $\mathcal{A}$ so in the top-level look-up is $q1$ mapped to the pointer to transition cluster. In this transition cluster is symbol a mapped to the set of states (in this case $(\{q_1, q_2\})$) which are accessible from q_1 under a. If one wants to insert a new transition $q_3 \xrightarrow{e} q_2$ to a NFA $\mathcal{C}$, the look-up table pointed by automaton $\mathcal{C}$ is duplicated and a state q_3 is inserted to it. The NFA $\mathcal{C}$ now points to that newly duplicated look-up table. State q_3 is in this look-up table mapped to pointer to the newly created transition cluster. Symbol e is inserted to this new transition cluster and mapped to the set of state which contains just state q_2.

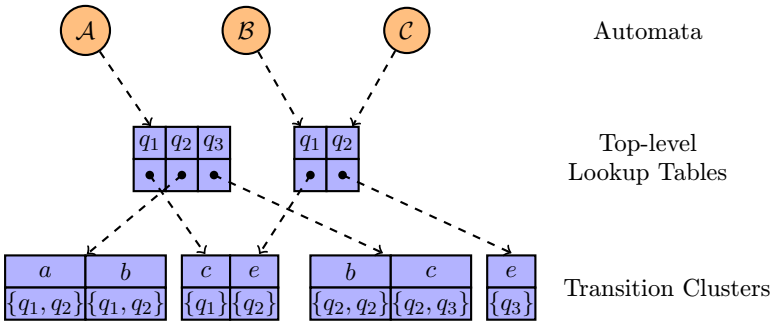


Figure 5.1: The data structure for storing transition of an finite automaton. There is a hash table (top-level look-up table) which map a state of a FA to the pointer to another hash table (transition cluster). Transition cluster maps a symbol of the input alphabet to a set of states.

5.2 Data Structure for Start and Final States

As it was mentioned in previous section 5.1.1 it is necessary to keep start and final states in the special sets to be able distinguish between them and the others states of an automaton. This is also main usage of these sets during operations over finite automata so there is no need to create special data structure and unordered set is efficient enough for this purposes.

5.3 Translation of the States and Symbols

An automaton is always parsed and converted to the intern representation from its input format. The conversion to the internal representation is based on mapping states from an input type (e.g. text description of the automaton) to the integers. This principle is also applied for symbols of the input alphabet. Mechanism of translation is illustrated by figure 5.2. The mechanism brings better efficiency for manipulation with states and symbols during the operations. It also provides unification of all input forms to the one internal representation.

Processing of some operations (e.g., union) can causes reindexing of the states what means that the integers which represents a state is changed. When the integer is changed an old value is mapped to a new one by a hash table what keeps relation between a original input value and the new integer value.

When the operations over a NFA are performed the NFA is often serialized back to the input format. The result automaton's states and symbols are mapped back to the input notation using hash tables where has been the mapping stored. This principle brings more readable output of serialization because the original notation is kept.

5.4 Usage of the Timbuk Format

VATA library provides possibility to load a finite automaton from a text specification. The text specification of NFA has to have a standard format but there is no such format for the finite automata so it was chosen the Timbuk format [17]. The Timbuk format is primarily used for description of the tree automata but can be also used for the finite automata after

q_1	q_2	q_3	q_4
1	2	3	4

a	b	c	d
1	2	3	4

Figure 5.2: The figure shows a principle of the translation of the input format to the internal representation by a hash table. The states (the left hash table) or of the symbols (the right hash table) of a NFA are mapped from strings to the integers.

some modifications. This format was also used as the input format of tree automata in VATA library.

Here is an example of an finite automaton defined by text description in the Timbuk format:

`Ops` $a : 1 \;\; x : 0$
`Automaton foo`
`States` $s \;\; p \;\; q \;\; f$
`Final States` f
`Transitions`
$x \to s$
$a(s) \to p$
$a(s) \to q$
$a(p) \to f$
$a(q) \to f$

On the first line of the specification in the Timbuk format is specified that the automaton has only one symbol of the input alphabet a with arity one (arity of the symbols of finite automata will be always one). The need of specification of the arity of an input symbol is a lack which comes from the original purpose of the Timbuk format because it is necessary to give the arity of an symbol of an input alphabet of an tree automaton.

The second symbol x with arity zero is not actually symbol of the input alphabet but is used for definition of the start states. The start states are defined in section *Transitions* by the transitions which has on the left side some symbol with zero arity and on the right side of the transition is a start state. This is again disadvantage of the Timbuk format because tree automata have no start states.

On the second line of the given example is name of the automaton (in our example is the name *foo*). On the third line is a list of states of the automaton and on the fourth line is a list of final states of the automaton.

Then there is list of the transitions of the automaton. For example, the transition $s \xrightarrow{a} q$ is in the Timbuk format described like $a(s) \to q$.

5.5 Algorithms for Basic Operations

In this section are described algorithms used for implementation of the basic operations like union, intersection or removing useless states and others.

5.5.1 Union

The union of two NFA $\mathcal{A}$ and $\mathcal{B}$ is described in section 2.2.3 and is done by following algorithm. First, a brand new automaton is created (this automaton will be result of union).

Sets of start and final states are copied to this automaton from both original automata. Then the all transitions from $\mathcal{A}$ and $\mathcal{B}$ are added to the newly created automaton. What is the most important during these operations is reindexing of states (it is supposed that the both automata have the same input alphabet so the symbols of it are not reindexed). The reindexing means that there is created index which maps integer that represents a state in the input automaton to a new integer which will represent the same state in the automaton created by this union.

The reindexing of states is done because the same integer can be used for representing one state of a NFA $\mathcal{A}$ and also another state of a NFA $\mathcal{B}$ and it is important to be able to distinguish between these two states in the result NFA. This technique also makes text output of serialization of the result automaton more readable because its states have the same names as it has in the input automata, only indices 1 and 2 are added to be able to distinguish between states of both automaton. E.g., a state q of NFA $\mathcal{A}$ and a state q of NFA $\mathcal{B}$ are in the result automaton denoted as q_1 and q_2.

Union of Disjunct States

The special case of union of two NFA is union of disjunct states of these NFA. This is done by copying of the first NFA to the result automaton and then the states (and transitions which contain these states) of the second NFA which are not already in the result NFA are copied to the result automaton. No reindexing of states is done during this operation.

5.5.2 Intersection

The intersection of two NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$ is defined in preliminaries 2.2.3. We define post-image of the product state $(p, q) \in \mathcal{A} \cap \mathcal{B}$ for a given symbol $a \in \Sigma$ by:
$Post_a((p, q)) := \{(p', q') \mid \exists a \in \Sigma : (p, a, p') \in \delta_a, (q, a, q') \in \delta_b\}$.
The algorithm for intersection is described by the algorithm 3.

The principle of this algorithm is following. The both input NFA are explored parallel and to the result automata are added product states. A product state consists two states each from different automaton which are accessible for the same string of input alphabet. The transitions of the result automaton contain also these product states.

5.5.3 Reverse

The reversion of an NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ is an NFA $\mathcal{A}_{rev} = (Q_{\mathcal{A}_{rev}}, \Sigma, \delta_{\mathcal{A}_{rev}}, I_{\mathcal{A}_{rev}}, F_{\mathcal{A}_{rev}})$ which is created just by changing start state's set by final state's set what is done by assigning $I_\mathcal{A}$ to $F_{\mathcal{A}_{rev}}$ and $F_\mathcal{A}$ to $I_{\mathcal{A}_{rev}}$ and reverting all transitions so e.g., transition $p \xrightarrow{x} q \in \delta_\mathcal{A}$ is added to $\delta_{\mathcal{A}_{rev}}$ in form $q \xrightarrow{a} p$. This principle is described by the algorithm 4

Algorithm 3: Algorithm for intersection of NFA

Input: NFA's $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$, $\mathcal{B} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$
Output: NFA $\mathcal{A} \cap \mathcal{B}$=$(Q_{\mathcal{A}\cap\mathcal{B}}, \Sigma, \delta_{\mathcal{A}\cap\mathcal{B}}, I_{\mathcal{A}\cap\mathcal{B}}, F_{\mathcal{A}\cap\mathcal{B}})$

```
Stack = ∅;
forall the (p_A, p_B) ∈ I_A × I_B do
    Add (p_A, p_B) to I_{A∩B};
    Push (p_A, p_B) on Stack;
    if (p_A ∈ F_A ∧ p_B ∈ F_B) then
        Add (p_A, p_B) to F_{A∩B}
while (Stack ≠ ∅) do
    Pick and remove a product-state (p_A, p_B) from Stack;
    forall the (q_A, q_B) ∈ Post_a(p_A, p_B) do
        if (q_A ∈ F_A ∧ q_B ∈ F_B) then
            Add (q_A, q_B) to F_{A∩B}
        Add (p_A, p_B) --a--> (q_A, q_B) to δ_{A∩B};
        Push (q_A, q_B) on Stack;
return NFA A ∩ B;
```

Algorithm 4: Algorithm for reverting of an NFA

Input: NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$
Output: NFA $\mathcal{A}_{rev} = (Q_{\mathcal{A}_{rev}}, \Sigma, \delta_{\mathcal{A}_{rev}}, I_{\mathcal{A}_{rev}}, F_{\mathcal{A}_{rev}})$

```
F_{A_rev} = I_A;
I_{A_rev} = F_A;
forall the (p, a, q) ∈ δ_A do
    Add (q, a, p) to δ_{A_rev};
return NFA A_rev;
```

5.5.4 Removing Unreachable States

Let the NFA $\mathcal{B}$ be created by removing all unreachable states from an NFA $\mathcal{A}$ (an unreachable state of an NFA was defined in chapter preliminaries 2.2.4). The algorithm for removing all unreachable states implemented in VATA library is described by algorithm 5.

The intuition behind the algorithm is following. The NFA $\mathcal{A}$ is explored from its start states and to the result automaton are added only states which are reachable from these start states for some word $w \in \Sigma^*$. At first, all reachable states are found and added to a special set. Then all transitions with a reachable state on left side are added to a result NFA $\mathcal{B}$. If a found reachable state is final state of $\mathcal{A}$ it is also added to set of final states

of $\mathcal{B}$. A set of start states is copied from NFA $\mathcal{A}$ to NFA $\mathcal{B}$

Algorithm 5: Algorithm for removing the unreachable states of NFA

Input: NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$
Output: NFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$

```
1  Reachable = I_A;
2  Stack = Reachable;
3  while (Stack ≠ ∅) do
4      Pick and remove a p from Stack;
5      forall the q ∈ {q' | ∃a ∈ Σ : (p, a, q') ∈ δ_A)} do
6          if (q ∉ Reachable) then
7              Push q on Stack;
8              Add q to Reachable;
9  I_B = I_A;
10 forall the p ∈ Reachable do
11     if p ∈ F_A then
12         Add p to F_B;
13     Add {(p, a, q) | ∃a ∈ Σ ∧ q ∈ Q_A : (p, a, q) ∈ δ_A} to δ_B;
14 return NFA B;
```

5.5.5 Removing Useless States

The useless state of an NFA was defined in preliminaries section 2.2.4. Removing of the useless states from an NFA $\mathcal{A}$ is done simply by removing all unreachable states of the NFA $\mathcal{A}$, then is the NFA $\mathcal{A}$ reverted and the unreachable states are removed also in this reverted automaton and finally is $\mathcal{A}$ reverted back to the originally direction. The NFA $\mathcal{A}$ does not contain any useless states after these operations.

5.5.6 Get Candidate

Get a candidate (word), also called get a witness, is operation over a NFA $\mathcal{A}$ which creates a NFA $\mathcal{B}$ which language $L(\mathcal{B})$ is subset of a language $L(\mathcal{A})$ of the NFA $\mathcal{A}$ and is also non-empty if $L(\mathcal{A})$ is non-empty too. The NFA $\mathcal{B}$ should have as little states and transitions as possible.

The operation for getting candidate is implemented by the algorithm 6. This algorithm copies a set of start states of $\mathcal{A}$ to a set of start states of $\mathcal{B}$ and also add the start states to a set of reachable states. Then all transitions from $\delta_{\mathcal{A}}$ containing on the left side a state from the set of reachable states are added to $\mathcal{B}$ and finally all successors of the currently reachable states are added to this set too. This is repeated until the whole NFA $\mathcal{A}$ is copied to the NFA $\mathcal{B}$ or a final state is not accessible from any reachable state.

Algorithm 6: Algorithm for getting witness in NFA

Input: NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$
Output: NFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$

```
1  I_B = I_A;
2  Reachable = I_A;
3  Stack = Reachable;
4  while (Stack ≠ ∅) do
5      Pick and remove a p from Stack;
6      forall the {(p, a, q) | ∃a ∈ Σ : (p, a, q) ∈ δ_A} do
7          if (q ∉ Reachable) then
8              Push q on Stack;
9              Add q to Reachable;
10         Insert (p, a, q) to δ_B;
11         if q ∈ F_A then
12             Add q to F_B;
13             return NFA B;
14 return NFA B;
```

Chapter 6

Implementation

This chapter provides description of the module of VATA library for the finite automata. Loading of an finite automaton to explicit encoding will be described first. Then a list of the used modules from the original VATA implementation is given and finally implementation of algorithms for checking language inclusion are covered.

6.1 Loading and Manipulation with Finite Automata in the Explicit Encoding

Loading of an finite automaton to the explicit is done by the class *ExplicitFiniteAut* what is the main class for representation of an finite automaton. This class has the data members that implements the data structure for explicit encoding of an finite automaton described in previous chapter 5 and implements also copy-on-write principle. It is possible to load the finite automaton from a data structure returned by parser or directly from a text format. The class also provides dumping the finite automaton back to the text format. It implements the operations for manipulation with an automaton like setting specific state as start or final one. The class *ExplicitFiniteAut* also ensures translation (mentioned in section 5.3) of the states and symbols to the internal representation of integers.

6.2 Used Modules of the VATA Library

There are some parts of VATA library which can be used also for development of the new extension for finite automata. In this section is given a list of modules which are efficient to use also for finite automata module of library.

Parser and Serializer

For loading automaton from a text specification is used module of VATA library called *parser* and for serializing back to the specification is used module called *serializer*. Because the same input format has been used for finite and tree automata (format is described in the section 5.4) it is possible to use the original parser and serializer which have been already implemented. The parser returns a data structure which generally describes a finite automaton. The data structure is further processed and converted to the data structure for explicit encoding of the finite automata.

When one wants to dump an automaton from the internal representation back to the text format, the automaton is converted to a data structure which is identical with the data structure returned by parser. Description of the automaton in this data structure is given to the serializer which dumps it to the output format.

Simulation

One of the operations over tree automata provided by the VATA library is computing simulation of an automaton. For computation of the simulation relation of an finite automaton is possible to use the existing implementation of this operation. The difference is in the conversion of an finite automaton into the Labeled Transition System (LTS) which needs to be implemented in part of library for finite automata.

Utilities

The original VATA library also provides a lot of utilities which are also useful for implementation of extension for the finite automata. These utilities provides classes for easier processing of finite automata. For example, the classes *TwoWayDict* and *TranslatorStrict* are uses for conversion of an finite automaton to the explicit encoding, the class *Antichain2Cv2* for representing of antichain in algorithm 3.1 and the class *AutDescription* for representing an automaton after the parsing.

The usage of the of this utilities speeds up the development of the new module for finite automata and also keeps the library more compact because no redundant code is produced.

6.3 Macrostate Cache

The sets of states (so-called macrostates) are compared in both mentioned algorithms (described in sections 3.1 and 3.2) for checking inclusion of languages of the two NFA, respectively some relation between the macrostates is checked. It is possible that there will be needed to check the same relation between the same two macrostates several times. In the case of antichains algorithm is possible that there is checked $(p_1, P) \sqsubseteq (q_1, Q)$ and then $(p_1, P) \sqsubseteq (q_2, Q)$, where p_1, q_1, q_2 are states of the first NFA and P and Q are sets of states of the second NFA. When the $(p_1, P) \sqsubseteq (q_2, Q)$ is being checked the relation between p_1 and q_2 is very easy to get because they are just two states, but checking relation between P and Q is very computationally demanding, because the macrostates could contain many of states, but it is also not necessary to check the relation again because the result has already been computed by checking $(p_1, P) \sqsubseteq (q_1, Q)$.

Similar situation could happen using the algorithm based on bisimulation up to congruence. There one wants to know all rewriting rules which are possible to use for computing $X{\downarrow_R}$ for some macrostate X and relation of visited pairs of macrostates R. Searching for usable rules is also very computationally demanding and it could be efficient to save all usable rewriting rules from R for given macrostate X.

According to these facts, so-called *Macrostate cache* has been implemented for improving the performance by storing the results of once computed relations of macrostates. The cache stores all macrostates which have been generated during exploring of a product NFA. Each macrostate is stored in cache only once so the macrostates are not manipulated alone but it is worked only with pointers to the macrostates in the cache what brings the advantage that it is not necessary to compare the whole macrostates but just pointers.

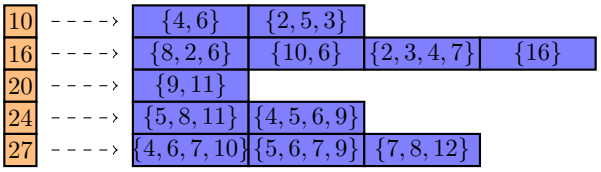


Figure 6.1: This figure shows the macrostate cache based on a hash table where key is the sum of a macrostate and a value is a list of the macrostates.

The macrostate cache is implemented as a hash table, where a key is the sum of the integers which represents the states of a macrostate and a value is a list of the macrostates which has the same state's sum. The macrostate cache can be seen on figure 6.1

6.4 Implementation of Antichain Algorithm

The implementation of the algorithm for checking language inclusion of NFA using antichains has been done by algorithm described in section 3.1. There were used data structures for representing antichains (classes *Antichain2Cv2* and *Antichain1c*) which were implemented for the modules for the tree automata.

The improvement of the antichain algorithm by using simulation is implemented too. This is done by parameterization of class for checking inclusion, where one of the given parameters is a relation which is simulation or identity (default).

Some optimization of this algorithm has been done during implementation and will be described in following subsections. For further subsections we suppose checking inclusion between a NFA $\mathcal{A}$ and a NFA $\mathcal{B}$.

6.4.1 Ordering of Antichain

The antichain algorithms keeps only the minimal set of the visited product states with respect to ordering given by $(r, R) \sqsubseteq (p, P)$ iff $p = r \wedge R \subseteq P$. The ordering $p \preceq r \wedge R \preceq^{\forall\exists} P$ is used for the optimization by simulation. Comparing (r, R) and (p, P) was implemented by one parameterized function for both orderings what is possible because $p = r$ is special case of $p \preceq r$ and the same holds for $R \subseteq P$ and $R \preceq^{\forall\exists} P$. But this implementation has shown as inefficient and the special implementation of this function for each ordering alone should be more efficient because $R \subseteq P$ could be decided without comparing both sets element by element when size of the macrostate R is greater then size of the macrostate P while in case of $R \preceq^{\forall\exists} P$ one element of macrostate P could simulate all elements of R so the size comparison is impossible. The ordering using simulation also needs to iterate through all visited product states to find all the elements p such that $p \preceq r$ what is not necessary in the case of basic version where is checked just $p = r$.

On the other side, the other optimization of simulation is based on the fact that if there is $p' \in P$ such that $p \preceq p'$ it is not needed to kept and further processed the product state (p, P). When one parameterized function is used for both versions of the algorithm, it causes unnecessary slowing down because the condition is always false in case of basic version of the algorithm. Although this optimization has not been already implemented, we suppose that the separation of the functions will be also efficiently used in this case.

6.4.2 Using Macrostate Cache

The antichain algorithm often checks whether $R \subseteq P$ or $R \preceq^{\forall\exists} P$ which can be both quite expansive operations and it can be efficient to store the results of these operations. Macrostate cache has been applied for this purpose so all used macrostates (such as R or P) are stored in this cache. Then it is possible to work just with pointers to the cache and what helps efficiently store relation between R and P. For example, the pointer to R is mapped to the pointer to P by a hash table when there is relation between R and P. There is also a hash table that maps the pointer to macrostate R to pointers to all macrostates which are not in relation with R.

6.4.3 Ordered Antichain

For checking inclusion of tree automata has not been used standard antichain, but for storing product states (p, P) which should be further processed is used ordered antichain [14] which prefers processing the elements with smaller size of macrostate P first. This optimization leads to reduction of produced states. The optimization has been implemented also for checking language inclusion of NFA and it reduces the number of produced product states too what brings the better performance.

6.5 Translation of an NFA to LTS

Before computing simulation relation over an NFA it is necessary to convert the NFA to labeled transition system (LTS), sort the states of the NFA to two or three partitions (final, non-final and class representing start state) and initialize the simulation relation. This is done by algorithm where all transitions of NFA are converted to the LTS and at the same time each of the processed states is sorted to the partitions in according if the state is final or not. If all states are final, there will be created only one partition in this part of the algorithm otherwise two partitions will be created. After all transitions are processed, there is added another one partition which represents start states. Then the simulation relation is initialized by the rules that each final state simulates other final state and each non-final simulates other non-final. The non-final one does not simulate final one, but final one simulates non-final one.

The created partitions, LTS and initialized simulation is given to the algorithm for computing the whole simulation.

6.6 Implementation of Bisimulation up to Congruence Algorithm

The algorithm for checking inclusion of the languages of NFA is described in its own section 3.2. For computation of a congruence closure, what is crucial part of the approach, was used an algorithm based on the rewriting rules (described 3.2.2). This algorithm was implemented generally for checking equivalence of NFA and its optimized version for checking inclusion was implemented too because the main goal of this work is to achieve the best performance of the inclusion checking. In this section will be described some implementation optimization of the algorithm. For this section let think two NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$.

6.6.1 Exploring Product NFA

Exploring of a product NFA $\mathcal{A} \times \mathcal{B}$ during checking inclusion between the languages of $\mathcal{A}$ and $\mathcal{B}$ could be done by the *breadth-first search* [12] algorithm or the *depth-first search* [12] algorithm what determines order in which the states of the product NFA are explored. Usage of one or another algorithm can effect the number of states that are processed. Difference between this two approaches is in usage of a data structure for storing of the newly generated states of $\mathcal{A} \times \mathcal{B}$. If the used data structure is list then breadth-first search is applied and if the used data structure is stack then depth-first search is applied.

The VATA library module for NFA currently supports only the breadth-first search algorithm. This approach has not been chosen for any special reason or superiority but has evolved during the implementation because the antichain algorithm uses for storing of the newly generated states also list what leads to the implementation of breadth-first search.

6.6.2 Using Macrostate cache

When one checks inclusion (or equivalence) of languages of NFA $\mathcal{A}$ and $\mathcal{B}$ there are generated states states of product NFA $\mathcal{A} \times \mathcal{B}$ which are pairs (X, Y) where X is macrostate of states of $\mathcal{A}$ and Y is macrostate of states of $\mathcal{B}$. X and Y are stored to macrostates cache and it is further worked only with pointers to this macrostates in cache.

The original algorithm does not control if the newly generated state (X, Y) has not already been visited and always checks if (X, Y) is in congruence closure of relation of the visited states R what is computationally demanding operation. Thanks to the working just with pointers to macrostates is easy to check whether the new state is already in a set of visited states without computing the congruence closure. It is done by hash table which maps pointer of a macrostate Y to list of all pointers to macrostates X such that $(X, Y) \in R$. Then it is easy to check whether a newly generated state has not already been processed.

This technique reduces the number of the states of $\mathcal{A} \times \mathcal{B}$ for which is necessary to compute congruence closure what helps improvement in the performance of the whole algorithm.

6.6.3 Computing Congruence Closure for Equivalence Checking

The computation of the congruence closure of the visited states of an NFA $\mathcal{A} \times \mathcal{B}$ for equivalence checking using rewriting rules as it was described in section 3.2.2 is computationally demanding operation, so there were implemented an optimization to enhance the performance of the algorithm.

The optimization is based on the observing that when the normal form of macrostates X and Y for some relation R are derived to find out if holds that $(X, Y) \in c(R)$ it is not necessary to use all possible rewriting rules of R and add as much states as possible to the normal form of macrostate. It is possible to stop derivation of $X{\downarrow_R}$ and $Y{\downarrow_R}$ when these sets are equal and it is also not necessary to achieve the equality $X{\downarrow_R}$ and $Y{\downarrow_R}$ by applying the same rules, not even by the same number of rules. This fact makes possible to control $X{\downarrow_R} = Y{\downarrow_R}$ on the fly and not only after the whole derivation.

This simplification leads to the implementation optimization, which is based on creating of $X{\downarrow_R}$ by applying all possible rewriting rules so it has as much states as possible. When a rule is applied during the derivation of $X{\downarrow_R}$ it is mapped in a hash table to the form of $X{\downarrow_R}$ after application of the rule. Then $Y{\downarrow_R}$ is derived gradually and after each step

is checked if the current form of $Y\downarrow_R$ is not same as any of the forms of $X\downarrow_R$ which has been reached during its derivation. But comparing $Y\downarrow_R$ after each step to all forms of $X\downarrow_R$ is not efficient and it slows down the algorithm instead of improving it. This leads to implementation where the form of $Y\downarrow_R$ after applying of a rewriting rule is compared to the form of $X\downarrow_R$ after the usage of the same rule. The second approach is not maybe so efficient because it is not detect if $X\downarrow_R = Y\downarrow_R$ as early as the first one but this disadvantage is compensated by lesser comparison of $X\downarrow_R$ and $Y\downarrow_R$.

6.6.4 Computing Congruence Closure for Inclusion Checking

In section 3.2.2 was described optimization that can be used when one uses the algorithm based on bisimulation up to congruence for checking the language inclusion. The optimization is based on the fact that inclusion checking is done by checking equivalence $\mathcal{A} \cup \mathcal{B} = \mathcal{B}$, so for a state (X, Y) holds $(X, Y) \in c(R)$ iff $X \subseteq Y\downarrow_R$ for a relation R of visited states. This optimization was implemented in the VATA library module for NFA to achieve the best performance in checking language inclusion.

This optimization is further enhanced by the following implementation's improvements. Once there is checked a generated product state (X, Y) and the normal form $Y\downarrow_R$ is computed it could be efficient to store all rewriting rules that was used during the computation because otherwise there can be generated another product state with Y in, e.g. (Z, Y) and the whole computation of $Y\downarrow_R$ has to be done again.

At the same time a rewriting rules can be used only in one direction $(Y \rightarrow X \cup Y)$ for a $(X, Y) \in R$. So when there is checked if a newly generated product state (P, Q) is in a congruence closure of R it is needed just to check if $Y \subseteq Q$ to apply the rewriting rule $Y \rightarrow X \cup Y$. If the rewriting rule is applied, macrostate X is added to $Q\downarrow_R$. Once the rewriting rule is possible to apply we know that it is possible to apply all elements of R containing Y so it is efficient to store that $Y \subseteq Q$.

This principle of storing applicable rules is implemented by a hash table where a pointer to a macrostate Q is mapped to a list of pointers to the macrostates where each of this macrostates Y enables to use all of the elements of relation R containing Y for computation of $Q\downarrow_R$. Notice that is not able to store elements of R which rewriting rule is not applicable because there are added gradually new elements to R and after applying of a new rewriting rule can be usable also rule which has not been usable last time.

During the experimental evaluation was found that this optimization is not as useful as it was expected because it does not happen very often that the one macrostate of NFA $\mathcal{B}$ (e.g. macrostate Q) is in the two different states of a product NFA $\mathcal{A} \cup \mathcal{B} \times \mathcal{B}$ (e.g., (P, Q) and (O, Q) where P and O are macrostates of $\mathcal{A} \cup \mathcal{B}$) and can slow down the algorithm because of the overhead given by checking if a normal form $Q\downarrow_R$ has not already been computed, so it was implemented a special function for computing normal form of a visited macrostate and special function for computing normal forms of macrostates which has not been already explored.

Chapter 7

Experimental Evaluation

This chapter is about the experimental evaluation of the algorithms for checking inclusion based on antichains and on the bisimulation up to congruence.

For both of the evaluations were used NFA for model checking provided by Dr. Lukáš Holík[1]. The evaluation was done on the set of about 40 000 of pairs of NFA.

The tests were performed on the server *merlin.fit.vutbr.cz* with CentOS 64bit Linux, 2× AMD Opteron (2,5 GHz, 4 cores, 12 MB cache) and 8 GB Ram.

7.1 Evaluation of Algorithm Based on Antichains

The implementation of the antichain algorithm is compared to the VATA library implementation of the antichain algorithm for tree automata. The tree automata's algorithms for checking language inclusion was tested in explicit encoding using upward direction and also downward direction. Timeout of computation was set to five seconds.

The comparison with the inclusion checking algorithm for tree automata in upward direction is given on the figure 7.4 where the whole data set is on the left plot and the right plot is focused. As it is written down in the table 7.3 the new implementation for NFA was faster in 98% of the test cases and in these cases was averagely two times faster. The tree automata's algorithm is faster only in two cases but in these cases is faster sixteen times. This acceleration of the tree automata's algorithm is some cases is not yet analyzed and could be object of further development.

The comparison of the new implementation of the algorithm for NFA with implementation for tree automata in downward direction using optimized cache is given on the figure 7.4 where the left plot again shows the whole data set and the right one shows focus on the time interval where the most of tests belong. The algorithm for NFA beats the tree automata's algorithm in the most cases (about 93%) and is about 211 times faster (all data are in table 7.3).

7.2 Evaluation of Algorithm Based on Bisimulation up to Congruence

The evaluation of the algorithm based on bisimulation up to congruence was done by implementation which includes all optimization described in section 6.6. There are provided

[1] Automata can be found on the web page: `http://www.fit.vutbr.cz/~holik/pub/ARMCautomata.tar.gz`

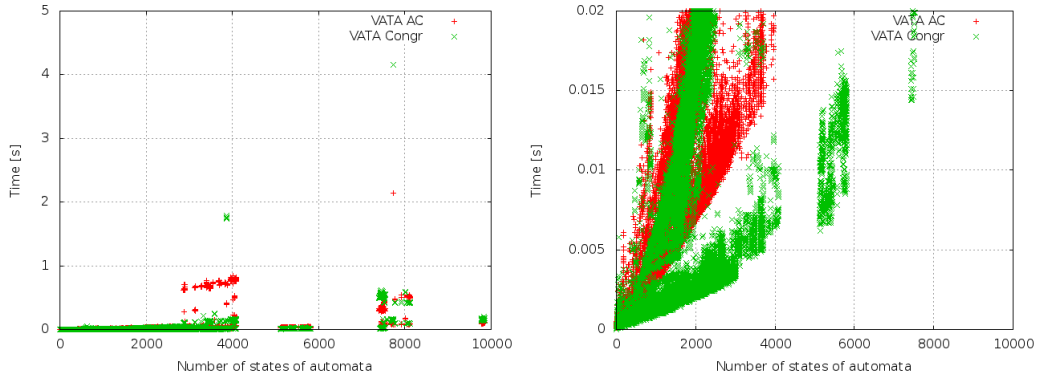


Figure 7.1: The comparison of VATA library implementation of antichain algorithm for tree automata in upward direction with VATA library implementation of the antichain algorithm for NFA.

	AC UP	**AC NFA**
winner	2%	98%
faster	15.91×	2.65×

	AC DOWN	**AC NFA**
winner	7%	93%
faster	10.78×	211.42×

Table 7.1: The left table shows comparison of VATA library for tree automata with checking inclusion in upward direction using the antichain algorithm with implementation of the antichain algorithm for NFA and the right table shows the same comparison but with for the downward direction version of the antichain tree automata's algorithm optimized by cache.

two comparison, the first one is with original OCaml implementation of this algorithm and the second one is with VATA library module for tree automata.

7.2.1 Comparison with OCaml Implementation

The algorithm based on bisimulation up to congruence was implemented [2] in *OCaml* language (object-oriented implementation of Caml language). This implementation provides checking of the equivalence and the inclusion of languages of NFA and also inclusion. It is also possible to use breadth-first search or depth-first search algorithm for searching product NFA and it is possible to use a simulation for improvement of performance of the algorithm.

For evaluation purposes was OCaml implementation run with breadth-first search (which is the only one currently implemented by VATA library), without simulation (which is not currently not provided by VATA library) and in the version for inclusion checking.

The comparison of the VATA library implementation and OCaml implementation can be seen on the figure 7.3. The plot shows the relation between time needed to check language inclusion and number of the states of the input NFA. The left plot shows the measurements on the whole data set and it is possible to see that the VATA library is especially faster for input automata with a lot of states. The right plot on the figure 7.3 is focused to the time interval where are the most of the measurements belong and shows that in some cases is

[2]The implementation can be found here: `http://perso.ens-lyon.fr/damien.pous/hknt/`

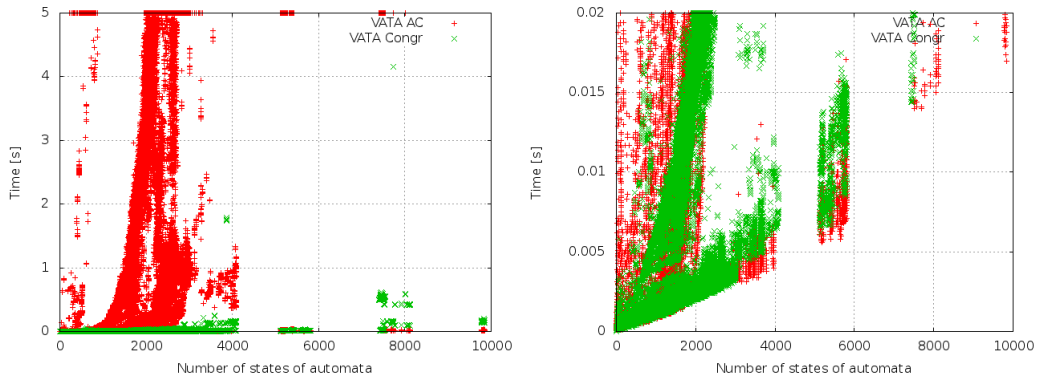


Figure 7.2: The figure shows comparison of VATA library implementation of the antichain algorithm for tree automata in downward direction using cache optimization with VATA library implementation of the antichain algorithm for NFA.

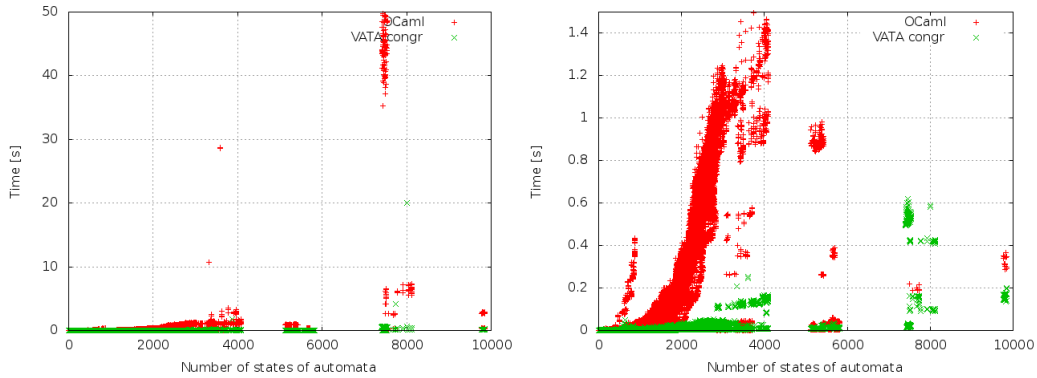


Figure 7.3: Comparison of OCaml implementation of a congruence algorithm and VATA library implementation of the algorithm.

OCaml implementation faster. In these cases only a few states were explored to check the language inclusion and the VATA library was slower due the overhead caused by its richer data structures (like macrostate cache). The plot also shows how time needed to check inclusion grows exponentially with number of states of NFA but the growth of amount of time is much faster in case of OCaml implementation.

The VATA library was faster in 92.5% of tested cases and in these cases were faster about 65 times, particular data are in the table 7.2.

7.2.2 Comparison with Tree Automata Implementation of VATA Library

It was also made evaluation which compares the VATA library for checking language inclusion of tree automata based on antichain algorithm. The VATA library for tree automata was used with explicit encoding and inclusion was checked in upward and also downward direction. For downward direction was used optimization based on a cache. The timeout for checking inclusion was set to 5 seconds.

	OCaml	VATA
winner	7.5%	92.5%
faster	6.43×	64.29×

Table 7.2: Table gives summarizing of the evaluation of comparing of OCaml implementation of a congruence algorithm and VATA library implementation of the algorithm.

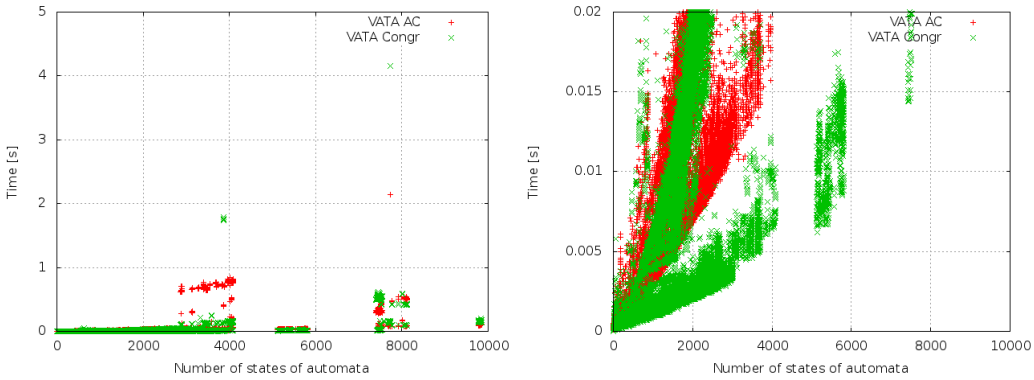


Figure 7.4: The comparison of VATA library implementation of the antichain algorithm for tree automata in upward direction with VATA library implementation of the congruence algorithm for NFA.

The result of comparison of the checking language inclusion using bisimulation up to congruence for NFA and antichains for tree automata is given on the figures 7.4 and 7.5, particular on the left plot in these figures. The plot shows number of states of the input automaton and time needed to check the inclusion. The new implementation for finite automata is faster in the most (about 95%) cases but is only about two times faster then the algorithm for upward direction. Checking language inclusion using downward direction is much slower than the congruence algorithm which is faster in 94% of cases and is faster for one hundred and sixty times. The particular data about speed up which brings implementation for NFA is given in table 7.3.

The figures 7.4 and 7.5 also show focus to a time interval where the most measurements belong. The figures show that VATA library for tree automata was also faster in some cases, what is caused by the fact the both approaches (antichain and bisimulation up to congruence) uses different attributes of relation of set of states that is necessary to check to verify if language inclusion holds.

	AC UP	CONGR
winner	5%	95%
faster	1.27×	2.34×

	AC DOWN	CONGR
winner	6%	94%
faster	1.90×	160.34×

Table 7.3: The left table shows comparison of VATA library for tree automata with checking inclusion upward with antichain algorithm with implementation of the algorithm based on bisimulation up to congruence and the right table shows the same comparison but with for the downward version of the antichain algorithm optimized by cache.

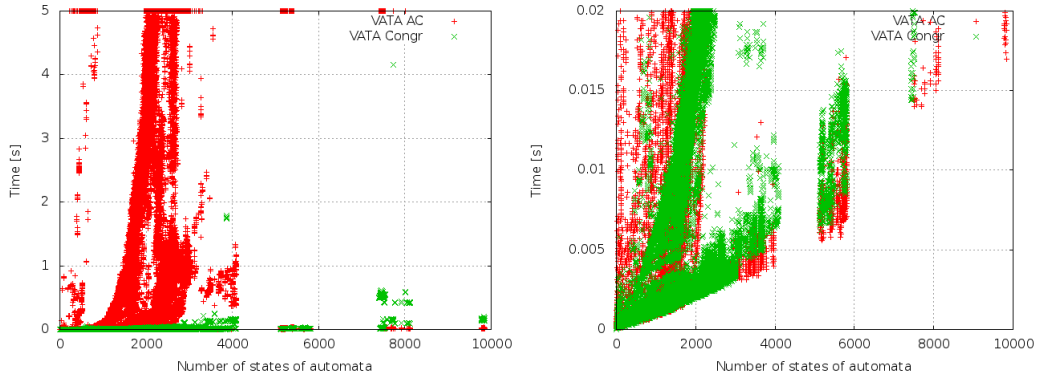


Figure 7.5: The figure shows comparison of VATA library implementation of the antichain algorithm for tree automata in downward direction using cache optimization with VATA library implementation of the congruence algorithm for NFA.

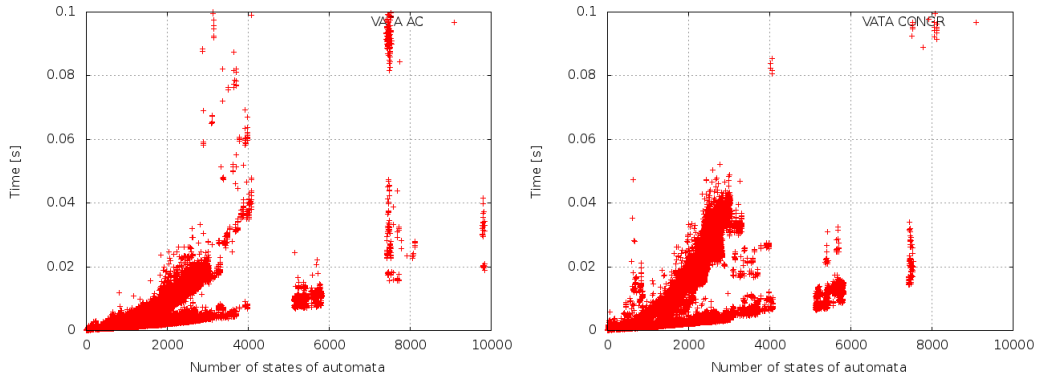


Figure 7.6: The comparison of VATA library implementation of the antichain algorithm for NFA (the left plot) with VATA library implementation of the congruence algorithm for NFA (the right plot).

7.3 Comparison of the algorithm for NFA

Finally, the both newly implemented algorithms for NFA, the one based on the antichains and the one based on bisimulation up to congruence, will be compared. The both algorithms were used in their optimized versions. The comparison is shown on the figure 7.6. As you can see, the results of the evaluations of both algorithms are very similar and the differences in the performance of them are very small. The results are summarized in the table 7.4. The antichain algorithm beats the congruence algorithm in 75% of tested cases. On the other side, the congruence algorithm is nearly four times faster in its winning cases than the antichain algorithm which is only 1.5 times faster in its winning cases. It is important notice that the results are dependent on the chosen test set because the both uses different attributes of the set of explored states of NFA during language inclusion checking.

	AC	**CONGR**
winner	76%	24%
faster	1.58×	3.74×

Table 7.4: Table shows result of comparison of the congruence algorithm and the antichain algorithm for NFA.

Chapter 8

Conclusion

The main goal of this thesis is to create an extension of the VATA library for the nondeterministic finite automata which are often used in the formal verification (e.g., model checking of safety temporal properties or abstract regular model checking) what is the branch of computer science which is the library focused on. The extension of the library is supporting basic operations like union, intersection, removing useless or unreachable states and etc., but the main aim of this work is to provide efficient implementation of the state-of-the-art algorithms for checking language inclusion of nondeterministic finite automata.

The new extension of VATA library uses the explicit encoding for representation of the finite automata. The data structures for the explicit encoding has been designed and implemented by modification and optimization of the data structures for the tree automata. The original VATA library has been analyzed to be able to determine which modules can be reused for a new extension and also to be able efficiently integrate the new extension.

To achieve the best performance for language inclusion checking are used the state-of-the-art algorithms, based on so-called antichains and so-called bisimulation up to congruence. The antichains algorithms is implemented in its default version and also optimized version which uses simulation of a finite automaton. The bisimulation up to congruence algorithm is implemented in its general version (for checking equivalence of finite automata) and also in version specialized to checking inclusion what brings better performance. The other improvement of this algorithm is realized by implementation's optimization.

For proving contribution of this work has been performed evaluation which compares the performance of our implementation of language inclusion checking to the other implementations. Our implementation beats the other tested implementation in over 90% of the tested cases. It faster about 100 times then the OCaml implementation of the algorithm for congruence closure and 2 times then the tree automata's algorithm.

The analysis of the cases where the tree automata's algorithm for language inclusion checking significantly beats the implementation specialized on NFA could be done for further optimization of the implementation. For the bisimulation up to congruence algorithm is possible to implement version of the algorithm which uses simulation for pruning out some other states which are not necessary to explore. The simulation is yet implemented for antichain algorithm for NFA but it has not been evaluated and optimized what can bring another improvement of performance. The full integration of the new extension of the VATA library could be done by application of it in the fields where the implementation of VATA library for tree automata is used.

Bibliography

[1] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When Simulation Meets Antichains: On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata. In *Proc. of TACAS 2010*, volume 6015, pages 158–174. Springer-Verlag, 2010.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. ISBN 978-0-262-02649-9.

[3] Nikos Baxevanis. Fare. `https://github.com/moodmosaic/Fare`, 2012 [cit. 2013-01-19].

[4] Filippo Bonchi and Damien Pous. Checking NFA Equivalence with Bisimulations up to Congruence. In *Proc. of POPL 2013*, pages 457–468. ACM, 2013.

[5] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract Regular Model Checking. In *Proc. of CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer Verlag, 2004.

[6] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-specific, Static Analyses. In *Proc. of PLDI 2002*, pages 69–82. ACM, 2002.

[7] Jesper G. Henriksen, Ole J.L. Jensen, Michael E. Jorgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders B. Sandholm. MONA: Monadic Second-Order Logic in Practice. In *Proc. of TACAS 1995*, volume 1019, pages 89–110. Springer Verlag, 1995.

[8] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *Proc. of FOCS 1995*, pages 453–462. IEEE Computer Society Washington, 1995.

[9] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Automata Theory, Languages, and Computation*. Pearson, 3rd edition, 2007. ISBN 0-321-47617-4.

[10] Stephan Kanthak and Hermann Ney. FSA: An Efficient and Flexible C++ Toolkit for Finite State Automata Using On-Demand Computation. In *ACL*, pages 510–517, 2004.

[11] Stephan Kanthak and Hermann Ney. The RWTH FSA Toolkit. `http://www-i6.informatik.rwth-aachen.de/~kanthak/fsa.html`, 2005 [cit. 2013-01-19].

[12] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 3rd edition, 1997. ISBN 0-201-89683-4.

[13] Dexter Kozen. *Automata and Computability*. Springer, 1997. ISBN 0-387-94907-0.

[14] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Proc. of TACAS 2012*, volume 7214, pages 79–94. Springer-Verlag, 2012.

[15] David Lutterkort. libfa. `http://augeas.net/libfa/index.html`, 2011 [cit. 2013-01-19].

[16] Anders Møller. dk.brics.automaton. `http://www.brics.dk/automaton/`, 2011 [cit. 2013-01-19].

[17] Web pages of Timbuk. Timbuk. `http://www.irisa.fr/celtique/genet/timbuk/`, 2012 [cit. 2013-01-29].

[18] M. De Wulf, L. Doyen, T.A. Henzinger, and J. F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proc. of CAV 2006*, volume 4144, pages 17–30. Springer-Verlag, 2006.

Appendix A

Storage Medium

The storage medium contains the sources of the VATA library including the new extension for finite automata. It also contains electronic version of this the text report.