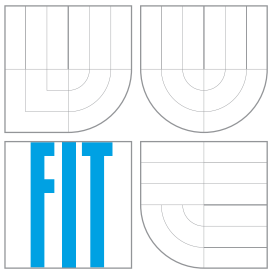
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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 formální verifikaci, při návrhu číslicových obvodů nebo 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 nedávné době bylo ale představeno několik nových přístupů, které právě explicitní determinizaci při testu jazykové inkluze předcházejí. Tyto přístupy využívají tzv. antichainů nebo tzv. bisimulace vzhůru ke kongruenci. Cílem 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, the design of digital circuits or 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 operation 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 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
13. května 2013

Poděkování

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

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. Chapter 3 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 a brief 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 the 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 a 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*.

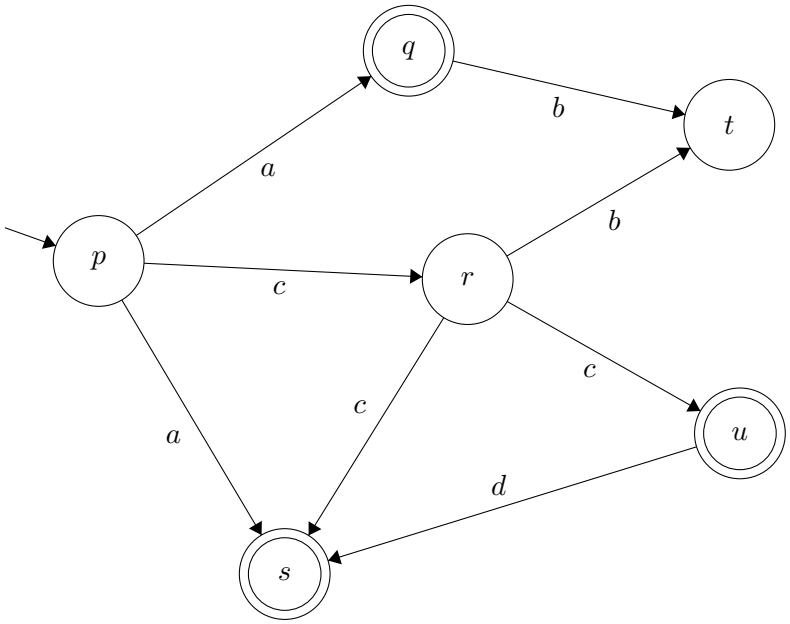


Figure 2.1: An example of an NFA

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

An example of an NFA over $\Sigma = \{a, b, c, d\}$ is shown in Figure 2.1. Notice the nondeterminism 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 Figure 2.2.

2.2.3 Run of a Finite Automaton

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

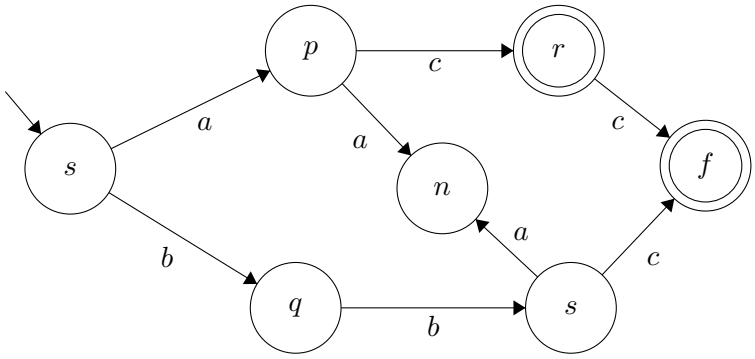


Figure 2.2: An example of an DFA

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

2.2.4 Language of a Finite Automaton

Consider an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$. The *language* of a state $q \in Q$ is defined as $L_{\mathcal{A}}(q) = \{w \in \Sigma^* \mid$ *there exists an accepting run of* $\mathcal{A}$ *from* q *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 $L_{\mathcal{A}} = L_{\mathcal{A}}(I)$. The language of NFA from Figure 2.1 is $L = \{a, cc, ccd\}$.

2.2.5 Complete DFA

Complete DFA $\mathcal{A}_{\mathcal{C}} = (Q_C, \Sigma, \delta_C, I_C, F_C)$ is the DFA where for any $p \in Q_C$, $a \in \Sigma$ exists $q \in Q_C$ such that $p \xrightarrow{a} q \in \delta_C$. It is possible to transform a DFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ to the complete DFA $\mathcal{A}_{\mathcal{C}} = (Q_C, \Sigma, \delta_C, I, F)$ such that $Q_C = Q \cup \{q\}$, $\delta_C = \delta \cup \{p \xrightarrow{a} q \mid \nexists r \in Q \,.\, p \xrightarrow{a} r \notin \delta_C\}$.

2.2.6 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

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

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

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

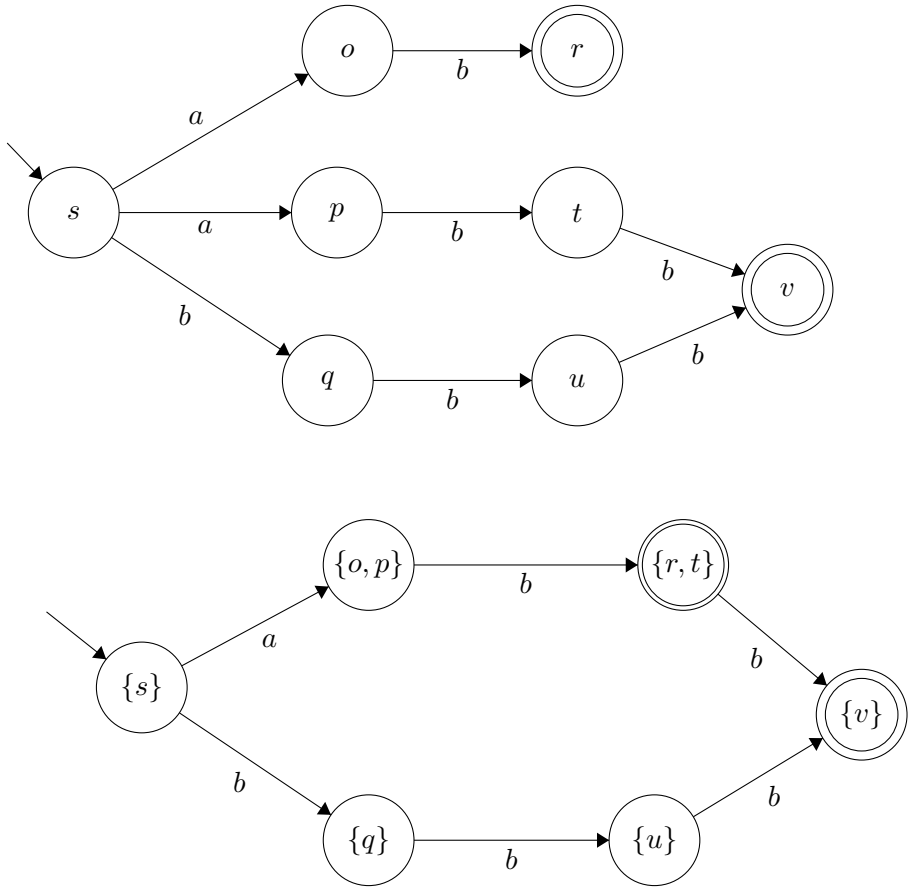


Figure 2.3: A simple example of NFA to DFA conversion via the subset construction with the reduction of inaccessible state. A small NFA with a small Σ is shown here, but for larger NFA could state explosion occur.

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 an equivalent DFA $\mathcal{A}_{det}$ for a given NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$. $\mathcal{A}_{det} = (2^Q, \Sigma, \delta_{det}, I, F_{det})$, where

- 2^Q is power set of Q, elements of 2^Q are called macrostates,
- $F_{det} = \{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\}$
- $\delta_{det}(Q', a) = \bigcup_{q \in Q'} \{r \in Q \mid q \xrightarrow{a} r \in \delta\}$.

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

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

The class of regular languages is closed under certain operation if the result of this operation on some regular language is always a regular language too.

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

- Union: $L = L_1 \cup L_2$. Union of two NFA is described in section 2.2.6.
- Intersection: $L = L_1 \cap L_2$. Intersection of two NFA is described in section 2.2.6.
- Complement: $L = \overline{L_1}$. Complement of NFA $\mathcal{A}$ is done by its determinizing (via subset construction described in section 2.2.6), transforming to complete DFA (via method described in 2.2.5) and swapping its final and non-final states set.
- Difference: $L = L_1 - L_2$. 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 $\mathcal{B}_{\mathcal{C}}$ (by methods 2.2.6 and 2.2.5) then complementing $\mathcal{B}_{\mathcal{C}}$ and finally creating intersection $\mathcal{A} \cap \overline{\mathcal{B}_{\mathcal{C}}}$.
- Reversal: $L = \{a_1 \dots a_n \in \Sigma^* \mid y = a_n \dots a_1 \in L\}$. Reversion of an NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ is NFA $\mathcal{A}_{rev} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}_{rev}}, F_{\mathcal{A}}, I_{\mathcal{A}})$ where $\delta_{\mathcal{A}_{rev}} = \{(q, a, p) \mid p, q \in Q_{\mathcal{A}}, a \in \Sigma \,.\, p \xrightarrow{a} q \in \delta_{\mathcal{A}}\}$.
- Iteration: L^*. Iteration of an NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ is NFA $\mathcal{A}^* = (Q_{\mathcal{A}}, \Sigma, \delta^*_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}} \cup I_{\mathcal{A}})$ where $\delta^*_{\mathcal{A}} = \delta_{\mathcal{A}} \cup \{q \xrightarrow{a} i \mid f \in F_{\mathcal{A}}, i \in I_{\mathcal{A}} \,.\, q \xrightarrow{a} f \in \delta_{\mathcal{A}}\}$
- Concatenation: $L \cdot K = \{x \cdot y \mid x \in L \wedge y \in K\}$. Concatenation of an NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ and an 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 $\delta = \delta_{\mathcal{A}} \cup \{q \xrightarrow{a} i \mid f \in F_{\mathcal{A}}, i \in I_{\mathcal{B}} \,.\, q \xrightarrow{a} f \in \delta_{\mathcal{A}}\}$

Chapter 3

Checking Inclusion 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 deciding whether $L_\mathcal{A} \subseteq L_\mathcal{B}$. This problem is PSPACE-complete [18]. The *textbook* algorithm for checking inclusion $L_\mathcal{A} \subseteq L_\mathcal{B}$ is based on the observation that this holds iff $L_\mathcal{A} \cap L_\mathcal{B} = \emptyset$ and works by first determinizing $\mathcal{B}$ (yielding the DFA $\mathcal{B}_{det}$ using the subset construction algorithm presented in section 2.2.6), complementing it (yielding $\overline{\mathcal{B}_{det}}$) and constructing the NFA $\mathcal{A} \cap \overline{\mathcal{B}_{det}}$ 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 (the so-called antichains [18], its optimization using simulation [1], and the 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 the following terms for the further description of the new techniques for the inclusion checking. We denote a product state of an NFA $\mathcal{A} \cap \mathcal{B}_{det}$ as a pair (p, P) of a state $p \in Q_\mathcal{A}$ and a macrostate $P \subseteq Q_{\mathcal{B}_{det}}$. We define the post-image of a product state (p, P) of an NFA $\mathcal{A} \cap \mathcal{B}$ by $Post(p, P) := \{(p', P') \mid \exists a \in \Sigma : p \xrightarrow{a} p' \in \delta_\mathcal{A}, P' = \{p'' \in Q_\mathcal{B} \mid \exists p''' \in P : p''' \xrightarrow{a} p'' \in \delta_\mathcal{B}\}\}$

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 an NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ is a relation $\preceq \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$ such that if $p \preceq r$ then (i) $p \in F_\mathcal{A} \Rightarrow r \in F_\mathcal{A}$ and (ii) for every transition $p \xrightarrow{a} p' \in \delta_\mathcal{A}$, there exists a transition $r \xrightarrow{a} r' \in \delta_\mathcal{A}$ such that $p' \preceq r'$ [8]. Note that simulation implies language inclusion, i.e., $p \preceq q \Rightarrow L_\mathcal{A}(p) \subseteq L_\mathcal{A}(q)$. For two macro-states P and R of a NFA is $R \preceq^{\forall\exists} P$ shorthand for $\forall r \in R \,.\, \exists p \in P : r \preceq p$. Macrostate P of a NFA is final if there exists $p \in P$ which is final state of this NFA. A product state (p, P) of a NFA $\mathcal{A} \cap \mathcal{B}_{det}$ is witness, if p is final in automaton $\mathcal{A}$ and P is not final in automaton $\mathcal{B}$.

3.1.1 Antichains Algorithm Description

The Antichains algorithm [18] described in pseudocode in Algorithm 1 starts searching for a final state of the product automaton $\mathcal{A} \cap \overline{\mathcal{B}_{det}}$ 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

Algorithm 1: Language inclusion checking with antichains and simulations

Input: NFAs $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$, $\mathcal{B} = (Q_\mathcal{B}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$.
A relation $\preceq$ over $\mathcal{A} \cup \mathcal{B}$ that imply language inclusion.
Output: TRUE if $\mathcal{L}_\mathcal{A} \subseteq \mathcal{L}_\mathcal{B}$. Otherwise FALSE.

```
1  if there is a witness product-state in {(i, I_B) | i ∈ I_A} then
2      return FALSE;
3  Processed:=∅;
4  Next:= {(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      foreach (p, P) ∈ {(r', Minimize(R')) | (r', R') ∈ Post(r, R)} do
8          if (p, P) is a witness 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;
```

derives new states along the product automaton transitions and inserts them to the set of pairs $Next$ for further processing. Once a product state from $Next$ is processed and it is moved to the set of visited pairs $Processed$. $Next$ and $Processed$ 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 a pair (p, P) generated and there is $(r, R) \in Next \cup Processed$ such that $(r, R) \sqsubseteq (p, P)$, we can skip (p, P) and not insert it to $Next$ 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 $(r, R) \in Next \cup Processed$ 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 at lines 11–14 in the pseudocode.

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 $p_2 \in P - \{p_1\}$, such that $p_1 \preceq p_2$. We can remove the state p_1 from macrostate P, because if $L_\mathcal{A}(P)$ does not contain the word then $L_\mathcal{A}(P - \{p_1\})$ does not contains it either. This optimization is applied by the function $Minimize$ at the lines 4 and 7 in the pseudocode.

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]. Given a set X and an n-ary operation O over X, an equivalence relation $\sim_R$ is a congruence if $\forall a_1, \ldots, a_n, b_1, \ldots, b_n \in X$:

$$a_1 \sim_R b_1, \ldots, a_n \sim_R b_n \Rightarrow O(a_1, \ldots, a_n) \sim_R O(b_1, \ldots, b_n)$$

Given an NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and a relation $R \subseteq Q_\mathcal{A} \times Q_\mathcal{A}$. R is bisimulation if both R and R^{-1} are simulations on the the NFA $\mathcal{A}$.

The presented 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 $L_{\mathcal{A}} \cup L_{\mathcal{B}} = L_{\mathcal{B}}$ iff $L_{\mathcal{A}} \subseteq L_{\mathcal{B}}$.

This approach is based on the computation of a *congruence closure* $c(R)$ for some binary relation R on the 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 tries to build bisimulation that relates $\mathcal{A}$ and $\mathcal{B}$. The algorithm 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 the product automaton $\mathcal{A}_{det} \cap \mathcal{B}_{det}$ (so-called product states) are the pairs $(P_{\mathcal{A}}, P_{\mathcal{B}})$ of a macrostate $P_{\mathcal{A}} \subseteq Q_{\mathcal{A}}$ and a 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 $\mathcal{A}$ iff $P_{\mathcal{B}}$ contains a final state of B so the product state is not final state of $\mathcal{A}_{det} \cap \mathcal{B}_{det}$.

The optimization brought by this algorithm is based on computing 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 in Algorithm 2. The operation post on a product state (P, Q) of a NFA $\mathcal{A} \cap \mathcal{B}$ used in the pseudocode we define by: $Post(P, Q) := \{(P', Q') \mid \exists a \in \Sigma : P' = \{p' \in Q_{\mathcal{B}} \mid \exists p \in P : p \xrightarrow{a} p' \in \delta_{\mathcal{B}}\}, Q' = \{q' \in Q_{\mathcal{B}} \mid \exists q \in Q : q \xrightarrow{a} q' \in \delta_{\mathcal{B}}\}\}$

Algorithm 2: Language equivalence checking with congruence

Input: NFAs $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$, $B = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$.
Output: TRUE, if $L_{\mathcal{A}} = L_{\mathcal{B}}$. 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          continue;
7      if ¬((X ∩ F_A ≠ ∅) ⇔ (Y ∩ F_B ≠ ∅)) then
8          return FALSE;
9      Add elements from Post(X, Y) to Next;
10     Add X,Y to Processed;
11 return TRUE;
```

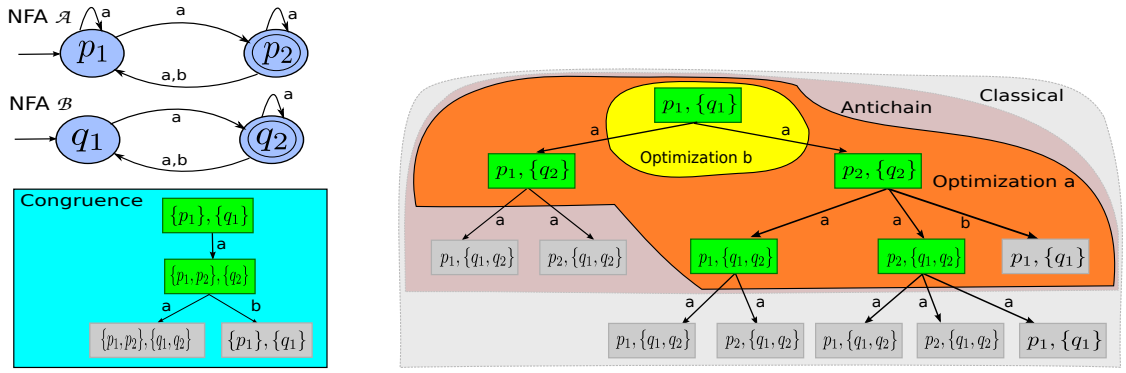


Figure 3.1: The figure is based on an example from [1]. It shows the procedure of checking language inclusion between two NFA using the mentioned approaches (which correspond to the labeled areas). The antichains algorithm reduces number of the generated states compared with the classical, e.g., $(p_2,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

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

3.2.2 Computation of Congruence Closure

The computation of the congruence closure is crucial for performance and efficiency of the whole method. The work described in this thesis implements an algorithm introduced in [4] which is based on the use of the so-called rewriting rules. For each pair of macrostates (X, Y) in a relation R of the visited macrostates there exist two rewriting rules which have 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 set is the set which is not changed after applying another rules. The normal form of a macrostate X created with the 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 in Figure 3.2). Consider 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 whether $(\{x, y\}, \{u\}) \in c(R)$ it is needed to compute the normal forms of the macrostates $\{x, y\}$ and $\{u\}$ A derivation of both normal forms is shown in Figure 3.3. The normal form of the set $\{x, y\}$ is derived in two steps. At the first step the rule $\{x\} \to \{x, u\}$ is applied (based on the pair $(\{x\}, \{u\}) \in R$) so we get a set $\{x, y, u\}$. As the second one, the rule $\{u\} \to \{y, z, u\}$ (based on the 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

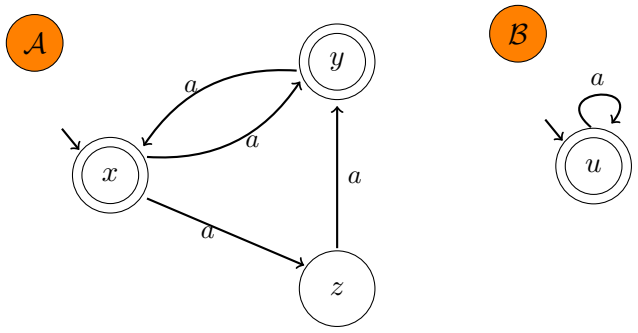


Figure 3.2: The figure shows two NFA $\mathcal{A}$, $\mathcal{B}$ which are used in the example describing computation of a congruence closure in Figure 3.3

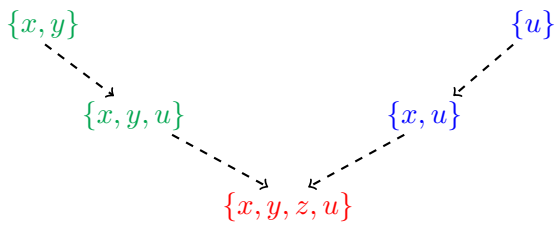


Figure 3.3: Derivation of the normal forms of the sets $\{x, y\}$ and $\{u\}$ using rewriting rules of the macrostates of a relation $R = \{(\{x\}, \{u\}), (\{y, z\}, \{u\})\}$.

steps too. At the first step, the rule $\{u\} \to \{x, u\}$ is applied so we get a set $\{x, u\}$ and then the 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 the product automaton $\mathcal{A} \times \mathcal{B}$ from this state.

A problem of this approach is that we do not know which rules of the relation R to use, in which order to use the and each rule can be used only once for computing the normal form. Due to this conditions the time complexity for finding an applicable rule is in the worst case $|R| \cdot |Q|$ where Q is a set of states of an NFA. The whole derivation of the normal set is bounded by the complexity $|R|^2 \cdot |Q|$ because we can apply at most 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 a macrostate (X, Y) is in the congruence closure of a relation R of the visited product states. The optimization is based on the fact that when one checks if the inclusion between NFA $\mathcal{A}$ and $\mathcal{B}$ holds it is done by checking if

$$\{x, y, u\} \quad \subseteq \quad \{x, y, z, u\} \quad \dashleftarrow \quad \{x, u\} \quad \dashleftarrow \quad \{u\}$$

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

$\mathcal{A} \cup \mathcal{B} = \mathcal{B}$ so in all product states (X, Y), X is a set of the 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 the following form [4]:

$$Y \to X \cup Y$$

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

As an example we give a computation of the congruence closure while checking inclusion between NFA $\mathcal{A}$ and $\mathcal{B}$ (both are in Figure 3.2). Let us have a relation of visited product states $R = \{(\{x, u\}, \{u\}), (\{y, z, u\}, \{u\})\}$ and a newly generated product state $(\{x, y, u\}, \{u\})$. The derivation of the normal form of the set $\{u\}$ is shown in Figure 3.4. The normal form of the set $\{u\}$ is derived in two steps, first the rule $\{u\} \to \{x, u\}$ is applied (based on the pair $(\{x, u\}, \{u\}) \in R$) so then we get the set $\{x, u\}$. Then the rule $\{u\} \to \{y, z, u\}$ (based on $(\{y, z, u\}, u)$) is used and the finally derived normal form is the 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. In this chapter, we will describe a few of the most prominent libraries. The described libraries are just examples which represent typical disadvantages of existing libraries like classical approach for language inclusion testing which needs the determinisation of a finite automaton.

In the second part of this chapter, the VATA library for *tree* automata will be introduced. The design of the library will be briefly described and also the operations for tree automata and the plans for an extension of the 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 7, 2011. The library can be downloaded and more information can be obtained from its webpage [16].

The library can use as the input a regular expression created by the Java *RegEx* class. It supports manipulation with NFA and DFA. Basic operations like union, intersection, complementation or membership test for a given word etc., are available.

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

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

libfa

libfa is a C library being part of the *Augeas* tool. The library is licensed under the LGPL, version 2 or later. It also support both versions of finite automata, NFA and DFA. Regular expressions may be used as an 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 this purpose.

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 the .NET framework. The library has the same characteristics as dk.brics.automaton or libfa and disadvantage in the need of determinisation is still here. Fare can be found on its webpage [3].

4.1.2 The RWHT FSA toolkit

RWHT FSA is a toolkit for manipulating finite automata described in [10]. The latest version is 0.9.4 from the 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. The library can be downloaded from [11].

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. The use of this technique leads to better memory efficiency.

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 the 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 the OCaml language for testing and evaluation purposes.

The algorithms using 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++ bringing greater efficiency compared to an OCaml implementation. A description of this library will be placed in the next section. Despite the fact that a C++ implementation could be more efficient than an OCaml implementation, there is currently no library or a toolkit similar to the VATA library providing an 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. The main application of VATA is in formal verification [14]. The 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/`

We define tree automata and related terms before the description of the VATA library itself. The *ranked* alphabet is a finite set of symbols with ranking function $\# : \Sigma \to \mathbb{N}$. A tree automaton (TA) is a quadruple $\mathcal{A} = (Q, \Sigma, \Delta, F)$, where Q is a finite set of states, Σ is a ranked alphabet, $F \subseteq Q$ is a set of final states and Δ is a set of transitions. A transition is a triple of the the form $((q_1, \ldots, q_n), a, q)$ where $q, q_1, \ldots, q_n \in Q, a \in \Sigma, \# = n$. The $((q_1, \ldots, q_n), a, q)$ can be denoted as $(q_1, \ldots, q_n) \xrightarrow{a} q$ (bottom-up representation of TA) or as $q \xrightarrow{a} q_1, \ldots, q_n$ (top-down representation of TA). Both notations are equivalent. Special cases are transitions where $n = 0$ which are called *leaf* transitions.

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 encodings are in the data structures for storing transition of tree automata. The semi-symbolic encoding is primarily for automata with large alphabets.

The main concept of the design of VATA library is shown in Figure 4.1 and we also give a brief description here. An input automaton is processed by one of the parsers (currently only Timbuk format parser is implemented). The result of parsing is a data structure with the general information about the automaton (the data structure stores a list of transitions of a given automaton, its final states, etc.). The main program chooses 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 the parser to the data structure used by the chosen encoding. The encodings also implement the operations over automata. When the automaton is processed it is possible to serialize it to an output format. This is done by one of the serializers (currently there is implemented only the Timbuk format serializer too) which takes as the input the same data structure which uses the parser.

As you can see in 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 the library such as parsers or serializers [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 the top-down direction (transitions are in the form $q \xrightarrow{a} (q_1, ..., q_n)$). The transitions are stored in a *hierarchical data structure based on hash tables*. The first level of the data structure is a hash table that maps states to *transition clusters*. These clusters are also look-up tables and map symbols of an input alphabet to sets of pointers (stored as *red-black trees*) to tuples of states. Storing tuples of states can be very memory demanding, so each tuple is stored only once and is referenced by different transitions. Inserting a new transition to this structure requires a constant number of steps (exception is the worst case scenario) [14]. This data structure can be seen in Figure 4.2.

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

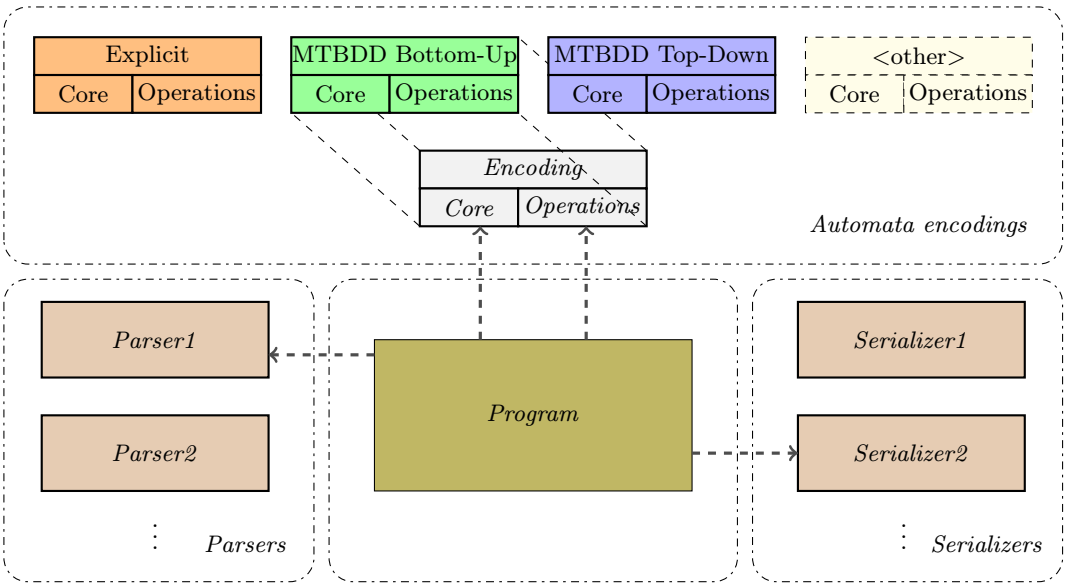


Figure 4.1: The VATA library design. The image is taken from [14]

.

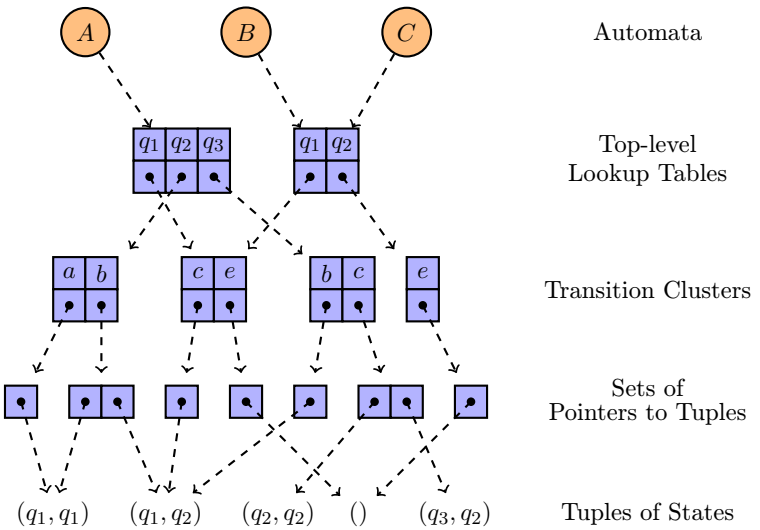


Figure 4.2: The data structure for storing transitions of the tree automaton. There is a hash table (top-level look-up table) which map a state to the pointer to another hash table (transition cluster). Transition cluster maps a symbols of input alphabet to the pointer to the set of pointers to the tuples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

Semi-symbolic Encoding

Transition functions in the semi-symbolic encoding are stored in *multi-terminal binary decision diagrams* (MTBDD), which are extension of *binary decision diagrams*. Two representations of tree automata are provided in the semi-symbolic encoding: top-down and bottom-up. The specific part is the saving of symbols into a MTBDD. In the top-down representation, the input symbols are stored in the MTBDD with their arity, because we need to be able to distinguish between two instances of the same symbols with a different arity. In the opposite case, the bottom-up representation does not need to store the arity, because it is possible to get it from the arity of tuples on the left-hand side of transition [14].

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

Operations

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

Optimized algorithms for inclusion testing based on the algorithms from [18, 1] are implemented. The inclusion test is implemented in more versions, so it is possible to use only some heuristics and compare different results.

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

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

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 a modular way, it is easy to extend it by another module. It was therefore decided not to create a brand new library but implement new extension of VATA for finite automata in C++ language.

The main goal is to provide an efficient implementation of the operation of checking language inclusion using state-of-the-art algorithms. To be precise the VATA library can be already used for finite automata which can be represented by unary tree automata. But the VATA library data structures for manipulating tree automata are designated for more complex data structures and new special implementation for finite automata will be definitely more efficient. Not only inclusion checking algorithms will be implemented,

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

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

but also algorithms for basic operations like union, intersection and 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 the parser and the serializer or computation of simulation over states of an automaton.

Chapter 5

Design

This chapter is primarily about the design of the newly created extension of the VATA library for finite automata. At first, the data structures used for storing a finite automaton will be described, then a principle of the translation of the states and the symbols of an NFA to the internal representation. The choice of the input format and its modification are justified. The algorithms for basic operations over NFA such as 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 the VATA library differ mainly in the 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 for the performance of the algorithms so it is important to take care when analyzing and designing it.

5.1.1 Analysis

A NFA is defined by the set of its states, its initial and final states (which are subsets of the set 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 initial and final states to be able to distinguish between them and the other states. However it is not necessary to store the whole set of states itself because it is given implicitly by the states used inside transitions. This also holds for the input alphabet of the NFA.

The transitions keep the most information about an NFA and are also often used during the operations over the NFA, so the performance of these operations strongly depend on the efficiency of the data structure used to store transitions. For an example, in many operations over the 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 automaton.

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 in 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 its modification also in the extension of the VATA library for finite automata.

5.1.2 Design of Data Structure for Transitions of NFA

The data structure for storing transitions of an 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. The described data structure is in Figure 5.1.

The data structure for storing transitions of an NFA is a simplification of the data structure for tree automata. Since transitions of a tree automaton has the 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 a symbol of the input alphabet of the tree automaton, and a finite automaton has transitions of the form: $q_1 \xrightarrow{a} q_2$ where q_1, q_2 are states of the finite automaton and a is a symbol of its alphabet, the simplification of the 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 a pointer to a tuple instead of the tuple itself.

In the case of finite automata this advantage disappears because there are no tuples of state but only states alone and keeping a pointer to one state would only bring unnecessary overhead (a size of a pointer to a state is usually equal or bigger than the state represented by an integer). This causes that it is not needed for the data structure for finite automata to use anything such as a set of pointers to tuples, but a set of states could be directly used instead of a set of pointers. The set of states would be referenced from transition clusters and would contain all states accessible from a given state under a certain symbol of the input alphabet.

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

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

This data structure also applies 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 the same and a new look-up table and a transition is created only when a new item is inserted to one of the automata. For example, NFA $\mathcal{B}$ and NFA $\mathcal{C}$ in Figure 5.1 are sharing the same data structure.

Let us give examples for searching and inserting a transition into this data structure for the NFA in Figure 5.1. If one wants to find all accessible states from the state q_1 over the symbol a in an NFA $\mathcal{A}$. First, the transition cluster that corresponds to q_1 is found in the top-level look-up table and then the set of states mapped by a is retrieved if there is such. If one wants to insert a new transition $q_3 \xrightarrow{e} q_2$ to an NFA $\mathcal{C}$, the look-up table for automaton $\mathcal{C}$ is duplicated because $\mathcal{C}$ has been sharing the look-up table with automaton $\mathcal{B}$, and finally a state q_3 is inserted into it. The NFA $\mathcal{C}$ now points to that newly duplicated

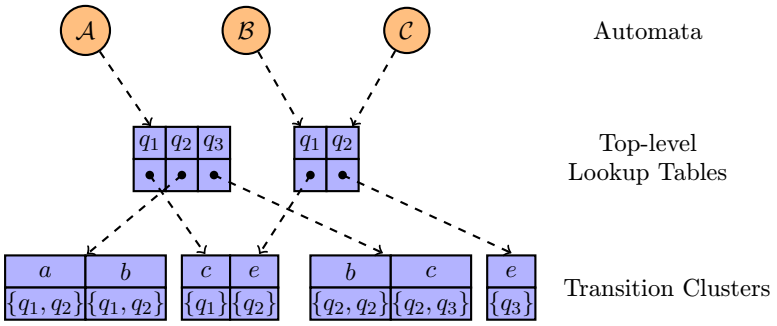


Figure 5.1: The data structure for storing transitions of an NFA. There is a hash table (top-level look-up table) which maps a state of an NFA to the pointer to another hash table (transition cluster). The transition cluster maps symbols of the input alphabet to sets of states.

look-up table. State q_3 is in this look-up table mapped to a pointer to the newly created transition cluster. The symbol e is inserted to this new transition cluster and mapped to the set of states which contains only the state q_2.

5.2 Data Structure for Initial and Final States

As it was mentioned in the previous section it is necessary to keep initial and final states in special sets to be able distinguish between them and the other states of an automaton. This is also the main use of these sets during operations over finite automata so there is no need to create a special data structure and a hash table is efficient enough for this purposes.

5.3 Translation of the States and Symbols

An automaton is always parsed and converted to the internal 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 integers. This principle is also applied for symbols of the input alphabet. The mechanism of translation is illustrated in 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.

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

When the operations over an NFA are performed it may be desirable to serialize the output NFA. The states and symbols of the resulting automaton are mapped back to the input notation using hash tables where the mapping has been stored. This principle brings more readable output of serialization because the original notation is kept (if it is possible).

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 an NFA are mapped from strings to the integers.

1. `Ops` $a:1\ \ x:0$
2. `Automaton foo`
3. `States` $s\ \ p\ \ q\ \ f$
4. `Final States` f
5. `Transitions`
6. $x \to s$
7. $a(s) \to p$
8. $a(s) \to q$
9. $a(p) \to f$
10. $a(q) \to f$

Figure 5.3: An NFA defined by text description in the Timbuk format

5.4 Use of the Timbuk Format

The VATA library provides the 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 a modification of the Timbuk format was chosen [17]. The Timbuk format is primarily used for the description of tree automata but can be also used for finite automata after some modifications. This format is also used as the input format of tree automata in the VATA library.

An example of a finite automaton defined by text description in the Timbuk format is given in Figure 5.3. On the first line in Figure 5.3 of the specification in the Timbuk format it 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 to specify of the arity of an input symbol is a necessity which comes from the original purpose of the Timbuk format because it is necessary to give the arity of a symbol of the input alphabet of the tree automaton.

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

On the second line in Figure 5.3 is the name of the automaton (our example the name is *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 a list of transitions of the automaton. For example, the transition $s \xrightarrow{a} q$ is in the Timbuk format described as $a(s) \to q$.

5.5 Algorithms for Basic Operations

In this section we describe algorithms used for implementation of basic operations, such as union, intersection or removing useless states.

5.5.1 Union

The union of two NFA $\mathcal{A}$ and $\mathcal{B}$ is described in section 2.2.6 and is done by the following algorithm. First, a brand new automaton is created (this automaton will be the result of the operation). Sets of initial and final states are copied to this automaton from both original automata. Then all transitions from $\mathcal{A}$ and $\mathcal{B}$ are added to the newly created automaton. What is the most important part during the previous operations is reindexing of states. The reindexing means that we create an index which maps integers that represent states in the input automaton to new integers representing 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 an NFA $\mathcal{A}$ and also another state of an 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 they have in the input automata, only indices 1 and 2 are added in order to be able to distinguish between states of both automata. E.g., a state q of the NFA $\mathcal{A}$ and a state q of the NFA $\mathcal{B}$ are in the result automaton denoted as q_1 and q_2.

Union of NFAs with Disjoint Sets of States

The special case of a union of two NFA is a union when states of these NFA have disjunct sets of states. This is done by copying 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 algorithm for computing the intersection automaton for two NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ and $\mathcal{B} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B})$ was introduced in Section 2.2.6. We define the post-image of the product state $(p, q) \in \mathcal{A} \cap \mathcal{B}$ for a given symbol $a \in \Sigma$ as: $Post_a((p, q)) := \{(p', q') \mid \exists a \in \Sigma : (p, a, p') \in \delta_a, (q, a, q') \in \delta_b\}$. The algorithm for intersection is given in Algorithm 3.

The principle of this algorithm is the following. Both input NFA are explored parallel and the product states are added into the result automaton. A product state consists of two states each from a different automaton which are accessible through the same word over the input alphabet. The transitions of the result automaton also contains these product states.

5.5.3 Reversal

The reversion of an NFA $\mathcal{A} = (Q_\mathcal{A}, \Sigma, \delta_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ is the NFA $\mathcal{A}_{rev} = (Q_\mathcal{A}, \Sigma, \delta_{\mathcal{A}_{rev}}, F_\mathcal{A}, I_\mathcal{A})$ which is created just by swapping the sets of initial and final states and reverting all transitions so e.g., transition $p \xrightarrow{x} q \in \delta_\mathcal{A}$ is added to $\delta_{\mathcal{A}_{rev}}$ in the form $q \xrightarrow{a} p$.

Algorithm 3: Algorithm for computing intersection of pair of NFA

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

```
1  Stack := ∅;
2  Reachable := ∅;
3  foreach (p_A, p_B) ∈ I_A × I_B do
4      Add (p_A, p_B) to I_{A∩B};
5      if (p_A, p_B) ∉ Reachable then
6          Add (p_A, p_B) to Reachable;
7          Push (p_A, p_B) on Stack;
8      if (p_A ∈ F_A ∧ p_B ∈ F_B) then
9          Add (p_A, p_B) to F_{A∩B}
10 while (Stack ≠ ∅) do
11     Pick and remove a product-state (p_A, p_B) from Stack;
12     foreach (q_A, q_B) ∈ Post_a(p_A, p_B), ∀a ∈ Σ do
13         if (q_A ∈ F_A ∧ q_B ∈ F_B) then
14             Add (q_A, q_B) to F_{A∩B}
15         Add (p_A, p_B) -a-> (q_A, q_B) to δ_{A∩B};
16         if (q_A, q_B) ∉ Reachable then
17             Add (q_A, q_B) to Reachable;
18             Push (q_A, q_B) on Stack;
19 return NFA A ∩ B=(Q_{A∩B}, Σ, δ_{A∩B}, I_{A∩B}, F_{A∩B});
```

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 section 2.2.3). The algorithm for removing all unreachable states implemented in the VATA library is described in Algorithm 4.

The intuition behind the algorithm is following. The NFA $\mathcal{A}$ is explored from its start states and to the result automaton there are added only those states which are reachable from these initial 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-hand side are added to the result NFA $\mathcal{B}$. If a found reachable state is a final state of $\mathcal{A}$ it is also added to the set of final states of $\mathcal{B}$. A set of initial states is copied from NFA $\mathcal{A}$ to NFA $\mathcal{B}$

5.5.5 Removing Useless States

Useless state of an NFA were defined in Section 2.2.3. Removing useless states from an NFA $\mathcal{A}$ is done simply by removing all unreachable states of the NFA $\mathcal{A}$, then we revert NFA $\mathcal{A}$ and remove unreachable states in this reverted automaton and finally $\mathcal{A}$ is reverted back to the originally direction. The NFA $\mathcal{A}$ does not contain any useless states after this sequence of operations.

Algorithm 4: Algorithm for removing the unreachable states of an 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 state p from Stack;
5      foreach q ∈ Post_a(p), ∀a ∈ Σ do
6          if q ∈ F_A then
7              Add q to F_B;
8          Add {(q -a-> q') | q' ∈ Q_A . ∃a ∈ Σ . q -a-> q') ∈ δ_A} to δ_B;
9          if (q ∉ Reachable) then
10             Push q on Stack;
11             Add q to Reachable;
12 I_B = I_A;
13 return NFA B = (Q_B, Σ, δ_B, I_B, F_B);
```

Algorithm 5: Algorithm for getting a witness in an 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 state p from Stack;
6      foreach p -a-> q ∈ {p -a-> q' | q' ∈ Q_A . ∃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             Remove useless states from NFA B;
14             return NFA B;
15 Remove useless states from NFA B;
16 return NFA B = (Q_B, Σ, δ_B, I_B, F_B);
```

5.5.6 Get Candidate

The operation of a getting a word (also called get a witness) of an NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ creates an NFA $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$ such that language $L_{\mathcal{B}}$ is a subset of the language $L_{\mathcal{A}}$ and is non-empty iff $L_{\mathcal{A}}$ is non-empty too. The NFA $\mathcal{B}$ should accept at most one word such that its length is the smallest from all words accepted by $\mathcal{A}$.

The operation for getting candidate is implemented in Algorithm 5. This algorithm

copies the set of initial states of $\mathcal{A}$ to a set of initial states of $\mathcal{B}$ and also adds the initial states to a set of reachable states. Then all transitions from $\delta_{\mathcal{A}}$ containing on the left-hand side a state from the set of reachable states are added to $\delta_{\mathcal{B}}$ and finally all successors of the currently reachable states are added to the set of reachable states too. This is repeated until final state is not accessible from any reachable state.

Chapter 6

Implementation

This chapter provides the description of the VATA library module for finite automata. Loading of a 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 the implementation of algorithms for checking language inclusion is covered.

6.1 Loading and Manipulation with Finite Automata in the Explicit Encoding

Loading of an finite automaton to the explicit representation is done by the class *ExplicitFiniteAut* which is the main class for representation of a finite automaton. This class has the data members that implement the data structure for explicit encoding of a finite automaton described in Chapter 5 and implements also the copy-on-write principle. It is possible to load the finite automaton from a data structure returned by a parser or directly from a text specification. The class also provides functions for serialization of the finite automaton back to the text specification. It implements the operations for manipulation with an automaton such as setting specific state as initial or final. The class *ExplicitFiniteAut* also ensures translation (mentioned in Section 5.3) of the states and symbols to the internal representation that uses integers.

6.2 Used Modules of the VATA Library

There are some parts of the VATA library which can be used also for the development of the new extension for the finite automata. In this section we give a list of modules which can be effectively used also for the finite automata module of the library.

Parser and Serializer

For loading an automaton from a text specification is VATA library uses a module called the *parser* and for serializing back to the specification the module called *serializer* is used. Because the same input format has been used for finite and tree automata (the format is described in Section 5.4), it is possible to use the original parser and serializer, which have already been 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 the explicit encoding of the finite automaton.

When one wants to serialize an automaton from the internal representation back to the text format, the automaton is converted to a data structure which is identical to the data structure used by the parser. The description of the automaton in this data structure is given to the serializer which transforms it into the output format.

Simulation

One of the operations over tree automata provided by the VATA library is computation of the maximum simulation relation over an NFA. 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 a finite automaton into the Labeled Transition System (LTS) which needs to be implemented in the part of library for finite automata.

Utilities

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

The use of these utilities sped up the development of the new module for finite automata and also kept the library more compact because no redundant code has been 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 two NFA, in particular some relation between the macrostates is checked. It is possible that it will be needed to check the same relation between the same two macrostates several times. In the case of the antichains algorithm it 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 other NFA. When the relation $(p_1, P) \sqsubseteq (q_2, Q)$ is being checked the relation between p_1 and q_2 is very easy to get because they are just two states, but checking the relation between P and Q is very computationally demanding, because the macrostates might contain many of states, but it is also not necessary to check the relation again because the result has already been computed while checking $(p_1, P) \sqsubseteq (q_1, Q)$.

A similar situation could happen using the algorithm based on bisimulation up to congruence. There one wants to find all rewriting rules which are possible to be used for computing $X{\downarrow_R}$ for a macrostate X and the 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 a given macrostate X.

According to these facts, the 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 the cache only once so the macrostates are not manipulated alone but it is worked only with pointers to the macrostates in the cache which brings the advantage that it is not necessary to compare the whole macrostates but just pointers.

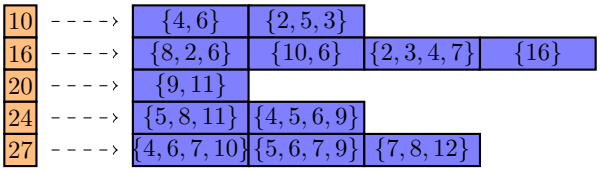


Figure 6.1: This figure shows the macrostate cache based on a hash table where the key is the sum of a macrostate and a value is a list of the macrostates.

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 sum of states. A hash function of states has also been used as the key of the hash table but it does not bring any improvements so the original implementation is kept. The macrostate cache can be seen in Figure 6.1.

6.4 Implementation of the Antichains Algorithm

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

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

Some optimization of this algorithm has been done during implementation and will be described in the following subsections. For further subsections we fix NFAs $\mathcal{A}$ and $\mathcal{B}$ and consider the problem $L_{\mathcal{A}} \subseteq L_{\mathcal{B}}$.

6.4.1 Ordering of an Antichain

The antichains algorithm keeps only the minimal set of the visited product states with respect to the 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 which is possible because $p = r$ is a 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 the size of the macrostate R is greater then size of the macrostate P while in the case of $R \preceq^{\forall\exists} P$ one element of the macrostate P could simulate all elements of R so the optimization cannot be used. The ordering using simulation also needs to iterate through all visited product states to find all the elements p such that $p \preceq r$ which is not necessary in the case of the basic version where just $p = r$ is checked.

On the other hand, 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 keep and further process the product state (p, P). When one parameterized function is used for both versions of the algorithm, it causes unnecessary slow down because the condition is always false in the case of the

basic version of the algorithm. Although this optimization has not been implemented yet, 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 expensive operations and it may be helpful to store the results of these operations. The 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 the pointers to the cache which helps to efficiently store the 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 the macrostate R to pointers to all macrostates which are not in relation with R.

6.4.3 Ordered Antichain

As the data structure for the $Next$ set we use an ordered antichain which prefers processing of elements with a smaller size of the macrostate P first. This optimization leads to the reduction of produced states. The optimization has been implemented also for checking language inclusion of NFA and it reduces the number of the produced product states too which yields a better performance.

6.5 Translation of an NFA into an LTS

Before computing the (maximum) simulation relation over an NFA it is necessary to convert the NFA into a LTS, sort the states of the NFA to two or three partitions (final, non-final and the class representing initial state) and initialize the simulation relation. This is done by an algorithm where all transitions of the input NFA are converted to edges of the LTS and at the same time each of the processed states is sorted to the partitions according to whether 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, another partition representing initial states is added. 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. A non-final one does not simulate a final one, but a final one simulates a non-final one. The created partitions, LTS and initialized simulation is given to the algorithm for computing the (maximum forward) simulation.

6.6 Implementation of the Bisimulation up to Congruence Algorithm

The algorithm for checking inclusion of the languages of NFA is described in Section 3.2. For computation of a congruence closure, which is a crucial part of the approach, we used an algorithm based on the rewriting rules (described in Section 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 we will describe a few implementation optimizations of the algorithm. For the rest of this section consider 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

Exploration of the product NFA $\mathcal{A} \cap \mathcal{B}$ while checking inclusion between the languages of the $\mathcal{A}$ and $\mathcal{B}$ could be done by the *breadth-first search* [12] algorithm or the *depth-first search* [12] algorithm, which determines the order in which the states of the product NFA are explored. The use of one or another algorithm can effect the number of states that are processed. The difference between these two approaches is in the use of a data structure for storing of the newly generated states of $\mathcal{A} \cap \mathcal{B}$. If the used data structure is a list then the breadth-first search is applied and if the used data structure is a 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 also uses a list for storing of the newly generated states which leads to the implementation of breadth-first search.

6.6.2 Using Macrostate cache

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

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

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

6.6.3 Computing Congruence Closure for Checking Equivalence

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

The optimization is based on the observation that when the normal forms of the macrostates X and Y for some relation R are derived to find out whether it 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 the macrostate. It is possible to stop the derivation of $X{\downarrow_R}$ and $Y{\downarrow_R}$ when these sets are equal and it is also not necessary to achieve the equality between $X{\downarrow_R}$ and $Y{\downarrow_R}$ by applying the same rules, not even by applying the same number of rules. This fact makes it possible to check $X{\downarrow_R} = Y{\downarrow_R}$ on the fly and not only after the whole derivation.

This simplification leads to the implementation optimization, which is based on creating $X\downarrow_R$ by applying all possible rewriting rules so that it has as many 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 it is checked whether the current form of $Y\downarrow_R$ is not the same as any of the forms of $X\downarrow_R$ which has been reached during its derivation. However 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 the 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 use of the same rule. The second approach is not maybe so efficient because it does not detect whether $X\downarrow_R = Y\downarrow_R$ as early as the first one but this disadvantage is compensated by a smaller number of comparisons of $X\downarrow_R$ and $Y\downarrow_R$.

6.6.4 Computing Congruence Closure for Inclusion Checking

In Section 3.2.2 an optimization was described that can be used when one uses the algorithm based on bisimulation up to congruence for checking language inclusion. The optimization is based on the fact that inclusion checking is done by checking the equivalence $\mathcal{A} \cup \mathcal{B} = \mathcal{B}$, so for a state (X, Y) it holds that $(X, Y) \in c(R)$ 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 improvements in the implementation. 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 were 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 rule can be used only in one direction $(Y \rightarrow X \cup Y)$ for $(X, Y) \in R$. So when it is checked if a newly generated product state (P, Q) is in the 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, the 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 the relation $Y \subseteq Q$.

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 the relation R containing Y for computation of $Q\downarrow_R$. Notice that it is not possible to store elements of R which rewriting rule is not applicable because there are added gradually new elements to R and after the application of a new rewriting rule can be usable also rule which could not be used the last time.

During the experimental evaluation it was found that this optimization is not as useful as it was expected because it does not happen very often that a single macrostate of NFA $\mathcal{B}$ (e.g. a macrostate Q) is in two different states of a product NFA $(\mathcal{A} \cup \mathcal{B}) \cap \mathcal{B}$ (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 as a special function for computing the normal form of a visited macrostate and a special function for computing the normal forms of macrostates which has not been already explored.

Chapter 7

Experimental Evaluation

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

For both of the evaluations we used NFA from abstract regular 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 Processors (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 algorithms for checking language inclusion of tree automata was tested in the explicit encoding using upward direction and also downward direction. The timeout of the computation was set to five seconds.

The comparison with the inclusion checking algorithm for tree automata in the upward direction is given in Figure 7.1 where the whole data set is in the left plot and the right plot is zoomed. The new implementation for NFA was faster in 98 % of the test cases and in these cases was on average twice as fast. The algorithm for tree automata is faster only in 2 % of cases but in these cases it is faster sixteen times. This acceleration of the algorithm for tree automata in some cases has not yet been analyzed and could be an object of further development.

The comparison of the new implementation of the algorithm for NFA with the implementation for tree automata in the downward direction using the optimized cache is given in Figure 7.2 where the left plot again shows the whole data set and the right one shows the zoom on the time interval where most of the tests belong. The algorithm for NFA beats the algorithm tree automata in the most cases (about 93 %) and is about 211 times faster (all data are in Table 7.1).

7.2 Evaluation of Algorithm Based on Bisimulation up to Congruence

The evaluation of the algorithm based on bisimulation up to congruence was done with the implementation which includes all optimizations described in Section 6.6. There are

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

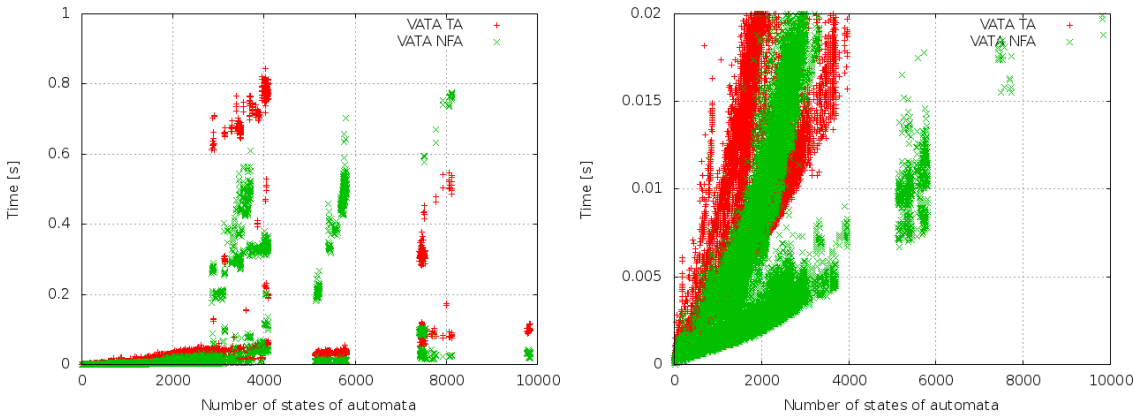


Figure 7.1: A comparison of the VATA library implementation of the antichains algorithm for tree automata in upward direction with the VATA library implementation of the antichains algorithm for NFA.

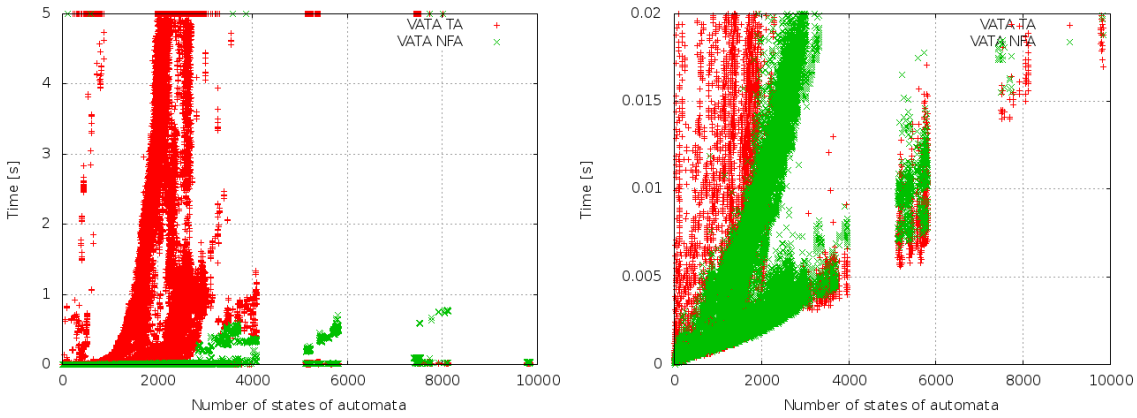


Figure 7.2: The figure shows a comparison of the VATA library implementation of the antichains algorithm for tree automata in the downward direction using the cache optimization with the VATA library implementation of the antichains algorithm for NFA.

	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 a comparison of the VATA library for tree automata with checking inclusion in the upward direction using the antichains algorithm with the implementation of the antichains algorithm for NFA and the right table shows the same comparison but with for the downward direction version of the antichains algorithm for tree automata optimized by a cache.

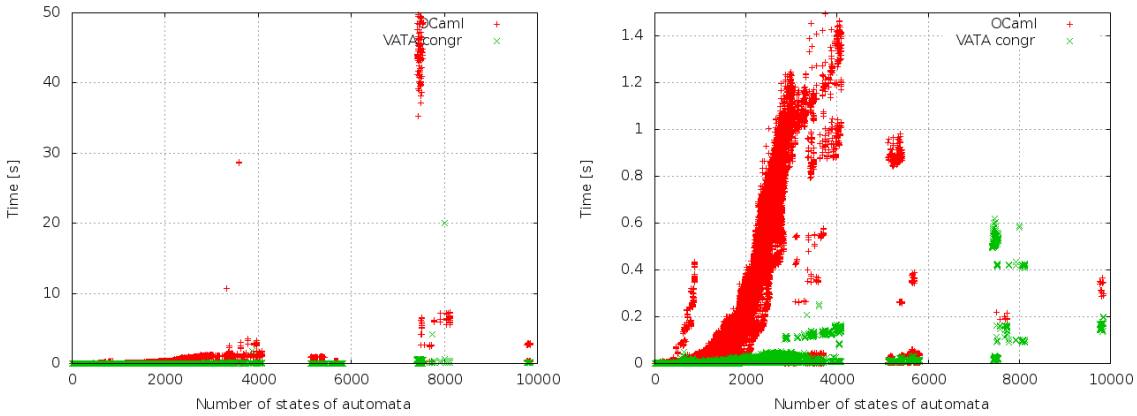


Figure 7.3: Comparison of the OCaml implementation of a congruence algorithm and the VATA library implementation of the algorithm.

two comparisons provided, the first one is with the original OCaml implementation of this algorithm [4] and the second one is with the 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 the *OCaml* language (an object-oriented implementation of the Caml language). This implementation provides checking of equivalence and inclusion of languages of NFA. It also allows to choose between the breadth-first search or the depth-first search algorithm for searching the product NFA and it is possible to use a simulation for an improvement of the performance of the algorithm.

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

The comparison of the VATA library implementation and the OCaml implementation can be seen in Figure 7.3. The plot shows the relation between the time needed to check language inclusion and number of 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 in Figure 7.3 is zoomed to the time interval where the most of the measurements belong and shows that in some cases the OCaml implementation is faster. In these cases only a few states were explored to check that the language inclusion and the VATA library was slower due to the overhead caused by its richer data structures (such as the macrostate cache). The plot also shows how the time needed to check inclusion grows exponentially with the number of states of NFA but the growth of the amount of time is much faster in the case of the OCaml implementation.

The VATA library was faster in 92.5 % of the tested cases and in these cases it was faster about 65 times. More detailed data can be found in Table 7.2.

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

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

Table 7.2: This table gives a summary of the evaluation of the performance of the OCaml implementation of a congruence algorithm and VATA library implementation of the same algorithm.

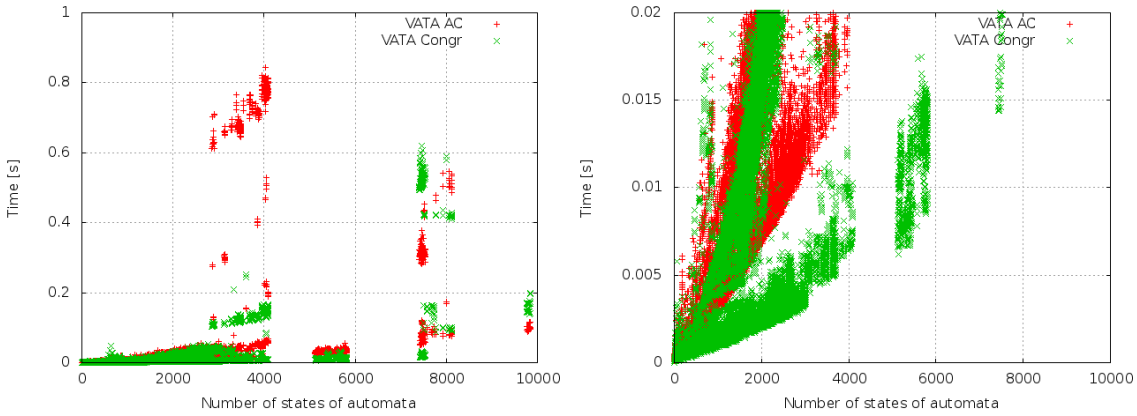


Figure 7.4: The comparison of the VATA library implementation of the antichains algorithm for tree automata in the upward direction with the VATA library implementation of the congruence algorithm for NFA.

7.2.2 Comparison with Tree Automata Implementation of VATA Library

We have also evaluated the performance of the inclusion checking algorithm based on bisimulation up to congruence with the VATA implementation of the antichains algorithm. The VATA library for tree automata was used with explicit encoding and inclusion was checked in the upward and also in the downward direction. For downward direction an optimization based on a cache was used. The timeout for checking inclusion was set to 5 seconds.

The results of the comparison of the checking language inclusion using bisimulation up to congruence for NFA and antichains for tree automata are given in Figures 7.4 and 7.5, where the left plots show the whole data set and the right plots are zoomed. The plots show the number of states of the input automata and the time needed to check the inclusion. The new implementation for finite automata is faster in the most (about 95 %) cases but is only about twice as fast as the algorithm for upward direction. Checking language inclusion using downward direction is much slower than the congruence algorithm which is faster in 94 % of the cases and is faster by the ratio of one hundred and sixty. The particular data about the speed-up of the implementation for NFA is given in Table 7.3.

Figures 7.4 and 7.5 also show zoom to a time interval where the most measurements belong. The figures show that the VATA library module for tree automata was also faster in some cases, which is caused by the fact the both approaches (antichain and bisimulation up to congruence) uses different attributes of the relation of sets of states that are necessary to check to verify that 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 a comparison of the VATA library for tree automata with checking inclusion upward using the antichains algorithm with the implementation of the algorithm based on bisimulation up to congruence, and the right table shows the same comparison but for the downward version of the antichains algorithm optimized by cache.

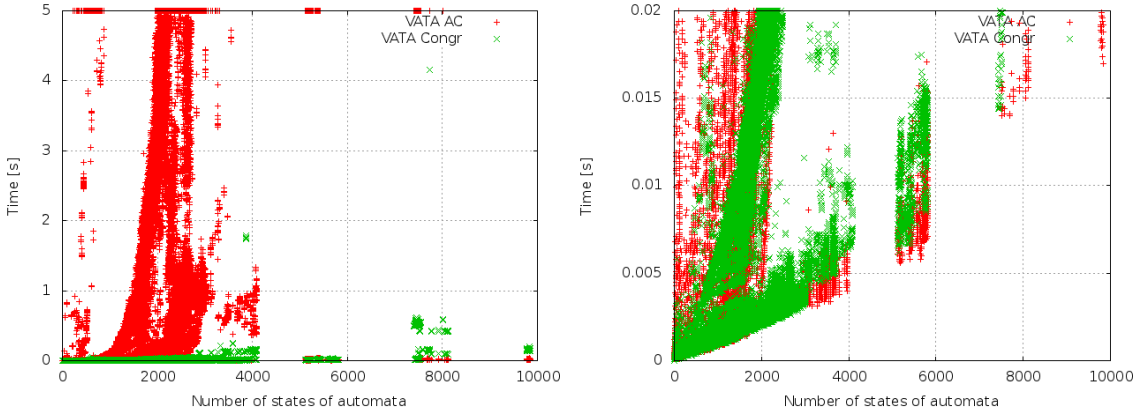


Figure 7.5: The figure shows a comparison of the VATA library implementation of the antichains algorithm for tree automata in the downward direction using the cache optimization with the VATA library implementation of the congruence algorithm for NFA.

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

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

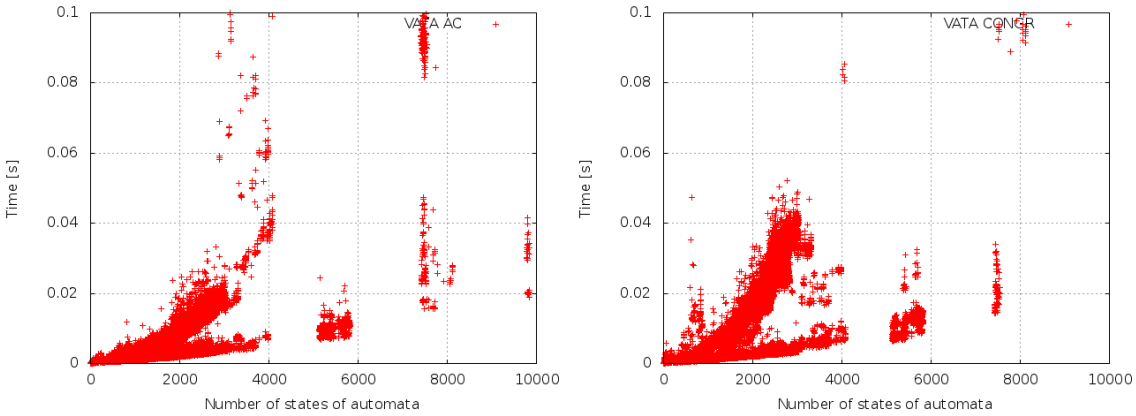


Figure 7.6: The comparison of the VATA library implementation of the antichains algorithm for NFA (the left plot) with the VATA library implementation of the congruence algorithm for NFA (the right plot).

7.3 Comparison of the Algorithms for NFA

Finally, both newly implemented algorithms for NFA, the one based on the antichains and the one based on the bisimulation up to congruence, will be compared. Both algorithms were used in their optimized versions. The comparison is shown in Figure 7.6. As you can see, the results of the evaluations of both algorithms are very similar and the differences in their performance are very small. The results are summarized in Table 7.4. The antichain algorithm beats the congruence algorithm in 76 % of the tested cases. On the other hand, the congruence algorithm is nearly four times faster in its winning cases than the antichains algorithm which is only 1.5 times faster in its winning cases. It is important to notice that the results depend on the chosen test set because both algorithms use different attributes of the set of explored states of the NFA while checking language inclusion.

Chapter 8

Conclusion

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

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

To achieve the best performance for language inclusion checking we use state-of-the-art algorithms, based on so-called antichains and so-called bisimulation up to congruence. The antichains algorithm was implemented in its default version and also in an optimized version which uses simulation over a finite automaton. The bisimulation up to congruence algorithm is implemented in its general version (for checking language equivalence of NFA) and also in the version specialized to checking inclusion. The other improvement of this algorithm is achieved by optimization of implementation.

An evaluation comparing the performance of our implementations and other implementations of checking language inclusion over NFA has been performed. Our implementation beats the other tested implementations in over 90 % of the tested cases. It is faster about 100 times than the OCaml implementation of the algorithm for congruence closure and twice as fast as the algorithm over tree automata.

A more detailed analysis of the cases where the algorithm for checking language inclusion over tree automata significantly beats the implementation specialized on NFA could be done for further optimization. For the bisimulation up to congruence algorithm it is possible to implement a version of the algorithm which uses simulation for pruning out some other states which are not necessary to explore. The simulation has already been implemented for antichains algorithm but it has not been evaluated and optimized which could bring another improvement in performance. Yet another interesting direction is to find suitable use cases of DFA for various task and evaluate whether they can improved by the use of NFA, in particular using the extension of the VATA library developed in this work.

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 an electronic version of this text report.