

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

MARTIN HRUŠKA

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EFEKTIVNÍ ALGORITMY PRO PRÁCI S KONEČNÝMI AUTOMATY

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

MARTIN HRUŠKA

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. ONDŘEJ LENGÁL

BRNO 2013

Abstrakt

Výtah (abstrakt) práce v českém jazyce.

Abstract

Výtah (abstrakt) práce v anglickém jazyce.

Klíčová slova

Klíčová slova v českém jazyce.

Keywords

Klíčová slova v anglickém jazyce.

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

April 28, 2013

Poděkování

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

© Martin Hruška, 2013.

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

Contents

1	Introduction	3
2	Preliminaries	5
2.1	Languages	5
2.2	Finite Automata	5
2.2.1	Nondeterministic Finite Automaton	5
2.2.2	Deterministic Finite Automaton	6
2.2.3	Operations over Finite Automata	6
2.2.4	Run of Finite Automaton	8
2.2.5	Minimum DFA	8
2.2.6	Language of Finite Automaton	9
2.3	Regular Languages	9
2.3.1	Closure Properties	9
3	Inclusion Checking over NFA	10
3.1	Checking Inclusion with Antichains and Simulation	10
3.1.1	Antichain Algorithm Description	10
3.2	Checking Inclusion with Bisimulation up to Congruence	11
3.2.1	Congruence Algorithm Description	12
4	Existing Finite Automata Libraries and the VATA Library	14
4.1	Existing Finite Automata Libraries	14
4.1.1	dk.brics.automaton	14
4.1.2	The RWHT FSA toolkit	15
4.1.3	Implementation of New Efficient Algorithms	15
4.2	VATA library	15
4.2.1	General	15
4.2.2	Design	16
4.2.3	Extension for Finite Automata	18
5	Design	20
5.1	Data Structures for Explicit Encoding of Finite Automata	20
5.1.1	Analysis	20
5.1.2	Design of Data Structure for Transtions of NFA	20
5.2	Start and final states data structure	22
5.3	Translation of the states and symbols	22
5.4	Usage of the Timbuk format	22
5.5	Algorithms for Basic Operations	23

5.5.1	Union	23
5.5.2	Intersection	24
5.5.3	Reverse	24
5.5.4	Removing Unreachable States	25
5.5.5	Removing Useless States	25
5.5.6	Get Candidate	25
6	Implementation	27
6.1	Loading and Manipulation with an Finite Automata in the Explicit Encoding	27
6.2	Used parts of existing implementation of VATA library	27
6.3	implementation of antichain algorithms	28
6.4	improvements for simulation	28
6.5	Implementation of Congruence Closure	28
7	Experimental evaluation	30
7.1	improvement given by congruence	30
8	Conclusion	31
8.1	further development	31
A	Storage Medium	34

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, designing of digital circuits or natural language processing. In formal verification alone are its uses abundant, for example in model checking of safety temporal properties, abstract regular model checking [5], static analysis [7], or decision procedures of some logics, such as Presburger arithmetic or weak monadic second-order theory of one successor (WS1S) [8].

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 second FA). The Classical (so called *textbook*) approach is based on *complementation* of the language of an 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 [6].

Recently, there has been a considerable advance in techniques for dealing with these problems. The new techniques are either based on the so-called *antichains* [6, 2] 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 [11], 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 checking language inclusion of NFA using bisimulation up to congruence (which is missing in VATA for tree automata).

After this introduction, in the 2nd chapter of this document, will be defined theoretical background. The 3rd chapter will describe efficient approaches to language inclusion testing. Existing libraries for finite automata manipulation and the VATA library will be introduced

in chapter 4. Design of extension for VATA will take place in chapter ?? . Implementation and optimization is possible to find in chapter 6. Evaluation will be described in chapter 7 and final conclusion in chapter 8.

Chapter 2

Preliminaries

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

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 $w = a_1 \dots a_n$, 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 $u = a_1 \dots a_2$ and $v = b_1 \dots b_n$ over Σ it holds that $\epsilon \cdot u = u \cdot \epsilon = u$ and $u \cdot v = a_1 \dots a_n b_1 \dots b_m$. We define a symbol Σ^* as a set of all words over Σ including the empty word and a symbol 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 by $L_1 \cdot L_2 = \{x \cdot y \mid x \in L_1, y \in L_2\}$. We define *iteration* L^* and *positive iteration* L^+ of a language L over an alphabet Σ *iteration* as:

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

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 is finite set of states, that $I \subseteq Q$. Elements of I are called initial states.
- F is finite set of states, that $F \subseteq Q$. Elements of F are called final states.

An example of an NFA is shown on the picture .

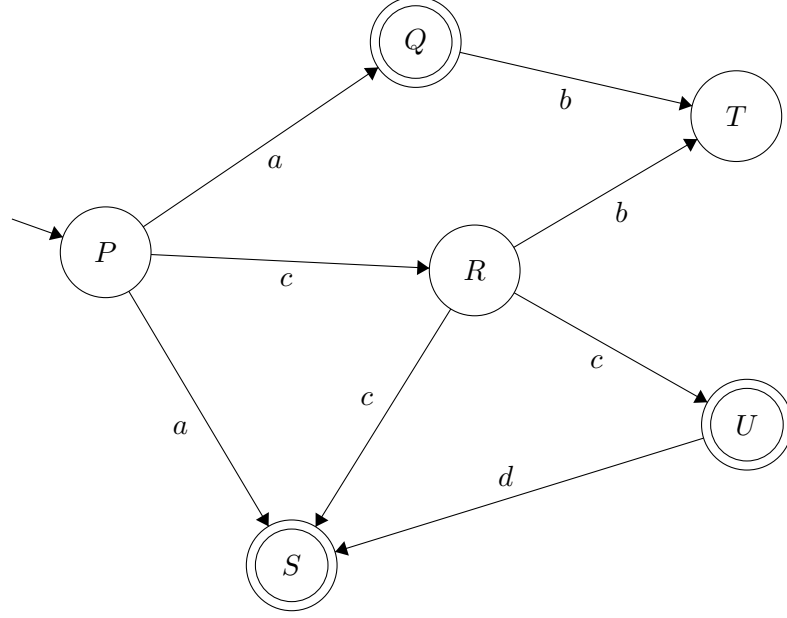


Figure 2.1: An example of a NFA

2.2.2 Deterministic Finite Automaton

A *deterministic finite automaton* (DFA) is a special case of an NFA, where δ is a partial function $\delta : Q \times \Sigma \rightarrow Q$ 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,
- $\delta: Q \times \Sigma \rightarrow Q$ 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, that $|I| \leq 1$.
- $F \subseteq Q$ is finite set of final states.

An example of a DFA is given on the picture 2.2.

2.2.3 Operations over Finite Automata

Automata Union

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

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

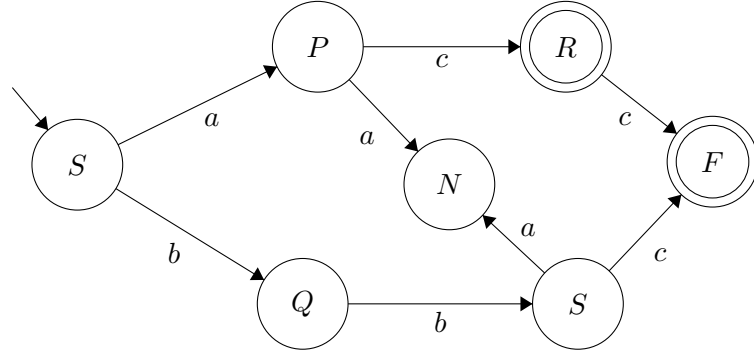


Figure 2.2: An example of a DFA

Automata Intersection

Given a pair of NFA, $A = (Q_A, \Sigma, \delta_A, I_A, F_A)$ and $B = (Q_B, \Sigma, \delta_B, I_B, F_B)$. Their intersection is defined by

$$A \cap B = (Q_A \cap Q_B, \Sigma, \delta, I_A \cap I_B, F_A \cap F_B)$$

where δ is defined by

$$\delta = \{(p_1, q_1) \xrightarrow{a} (p_2, q_2) \mid p_1 \xrightarrow{a} p_2 \in \delta_A \wedge q_1 \xrightarrow{a} q_2 \in \delta_B\}$$

Automata Product

Given a pair of NFA, $A = (Q_A, \Sigma, \delta_A, I_A, F_A)$ and $B = (Q_B, \Sigma, \delta_B, I_B, F_B)$. Their product is defined by

$$A \times B = (Q_A \times Q_B, \Sigma, \delta, I_A \times I_B, F_A \times F_B)$$

where δ is defined by

$$\delta = \{(p_1, q_1) \xrightarrow{a} (p_2, q_2) \mid p_1 \xrightarrow{a} p_2 \in \delta_A \wedge q_1 \xrightarrow{a} q_2 \in \delta_B\}$$

Subset construction

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

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

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

This classical („textbook“) approach is called *subset construction*. An example of this approach is shown on the picture 2.3.

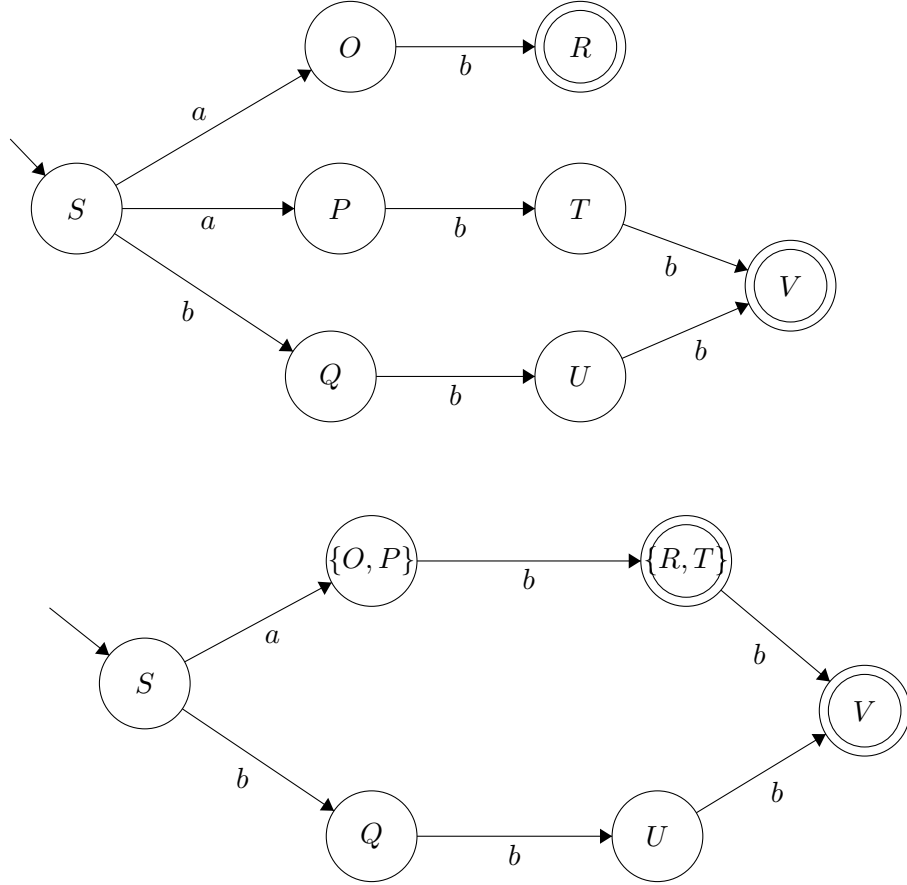


Figure 2.3: A simple example of NFA to DFA conversion via the subset construction. Here is shown small NFA with small Σ , but for larger NFA could state explosion occur.

2.2.4 Run of Finite Automaton

A *run* of an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ from a state q over a word $w = a_1 \dots a_n$ is a sequence $r = q_0 \dots q_n$, where $\forall 0 \leq i \leq n . q_i \in Q$ such that $q_0 = q$ and $(q_i, a_{i+1}, q_{i+1}) \in \delta$. The run r is called *accepting* iff $q_n \in F$. A word $w \in \Sigma^*$ is called *accepting*, if there exists an *accepting* run for w . An *unreachable* state q of an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is a state for which there is no run $r = q_0 \dots q$ of \mathcal{A} over a word $w \in \Sigma^*$ such that $q_0 \in I$. An *useless* (also called nonterminating) state q of an NFA $A = (Q, \Sigma, \delta, I, F)$ is state that there is no run $r = q \dots q_n$ of A over a word $w \in \Sigma^*$ such that $q_n \in F$. Given a pair of states p, q of an NFA $A = (Q, \Sigma, \delta, I, F)$, these states are equivalent if $\forall w \in \Sigma^* : \text{Run from } p \text{ over } w \text{ is accepting} \Leftrightarrow \text{Run from } q \text{ over } w \text{ is accepting}$.

2.2.5 Minimum DFA

Definition 2.2.1. *Minimum DFA satisfies this conditions:*

- *There are no unreachable states*
- *There is maximal one nonterminating state, which terminates on itself for each symbol.*

- *Equivalent states are collapsed.*

2.2.6 Language of Finite Automaton

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

2.3 Regular Languages

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

2.3.1 Closure Properties

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

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

- Union: $L_1 \cup L_2$
- Intersection: $L_1 \cap L_2$
- Complement: \overline{L}
- Difference: $L_1 - L_2$
- Reversal: $\{a_1 \dots a_n \in L \mid y = a_n \dots a_1 \in L\}$
- Iteration: L^*
- Concatenation: $L \cdot K = \{x \cdot y \mid x \in L \wedge y \in K\}$

Chapter 3

Inclusion Checking over NFA

Given a pair of NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$, the *language inclusion problem* is decision whether $L_{\mathcal{A}} \subseteq L_{\mathcal{B}}$ what is defined by standard set operations as $L_{\mathcal{A}} \cap \overline{L_{\mathcal{B}}} = \emptyset$. This problem is PSPACE-complete [6]. The *textbook* algorithm for checking inclusion $L_{\mathcal{A}} \subseteq L_{\mathcal{B}}$ works by first determinizing \mathcal{B} (yielding the DFA \mathcal{B}_{det} using subset construction algorithm 2.2.3), complementing it ($\overline{\mathcal{B}_{det}}$) and constructing the NFA $\mathcal{A} \times \overline{\mathcal{B}_{det}}$ accepting the intersection of $L_{\mathcal{A}}$ and $\overline{L_{\mathcal{B}_{det}}}$ and checking whether its language is nonempty. Any accepting run in this automaton may serve as a witness that the inclusion between \mathcal{A} and \mathcal{B} does not hold. Some recently introduced approaches (so-called antichains [6], its optimization using simulation [2] and so-called bisimulation up to congruence [4]) avoid the explicit construction of $\overline{\mathcal{B}_{det}}$ and the related state explosion in many cases.

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

3.1 Checking Inclusion with Antichains and Simulation

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

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

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

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

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

3.1.1 Antichain Algorithm Description

The antichains algorithm [6] starts searching for a final state of the automaton $\mathcal{A} \times \overline{\mathcal{B}_{det}}$ while pruning out the states which are not necessary to explore. \mathcal{A} is explored nondeterministically and \mathcal{B} is gradually determinized, so the algorithm explores pairs (p, P) . The antichains algorithm derives new states along the product automaton transitions and inserts them to the set of visited pairs X . X keeps only minimal elements with respect to

the ordering given by $(r, R) \sqsubseteq (p, P)$ iff $r = p \wedge R \subseteq P$. If there is generated a pair (p, P) and there is $(r, R) \in X$ such that $(r, R) \sqsubseteq (p, P)$, we can skip (p, P) and not insert it to X for further search.

An improvement of the antichains algorithm using simulation [2] is based on the following optimization. We can stop the search from a pair (p, P) if either (a) there exists some already visited pair $(r, R) \in X$ such that $p \preceq r \wedge R \preceq^{\forall\exists} P$, or (b) there is $p' \in P$ such that $p \preceq p'$. This first optimization is in algorithm 1 at lines 11–14.

Another optimization [2] of the antichain algorithm is based on the fact that $L_{\mathcal{A}}(P) = L_{\mathcal{A}}(P - \{p_1\})$ if there exists $p_2 \in P$, such as $p_1 \preceq p_2$. We can remove the state p_1 from macrostate P , because if $L_{\mathcal{A}}(P)$ rejects the word then $L_{\mathcal{A}}(P - \{p_1\})$ rejects this word too. This optimization is applied by the function *Minimize* at the lines 4 and 7 in the algorithm 1.

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

Algorithm 1: Language inclusion checking with antichains and simulations

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

```

1 if there is an accepting product-state in  $\{(s, S_B) \mid s \in S_A\}$  then
2    $\perp$  return FALSE;
3 Processed:= $\emptyset$ ;
4 Next:= Initialize( $\{(s, \text{Minimize}(S_B)) \mid s \in S_A\}$ );
5 while (Next  $\neq \emptyset$ ) do
6   Pick and remove a product-state  $(r, R)$  from Next and move it to Processed;
7   forall the  $(p, P) \in \{(r', \text{Minimize}(R')) \mid (r', R') \in \text{Post}((r, R))\}$  do
8     if  $(p, P)$  is an accepting product-state then
9       return FALSE;
10    else
11      if  $\nexists p' \in P$  s.t.  $p \preceq p'$  then
12        if  $\nexists (x, X) \in \text{Processed} \cup \text{Next}$  s.t.  $p \preceq x \wedge X \preceq^{\forall\exists} P$  then
13          Remove all  $(x, X)$  from  $\text{Processed} \cup \text{Next}$  s.t.  $x \preceq p \wedge P \preceq^{\forall\exists} X$ ;
14          Add  $(p, P)$  to Next;
15 return TRUE;
```

3.2 Checking Inclusion with Bisimulation up to Congruence

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

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

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

This technique was originally developed for checking equivalence of languages of automata but it can also be used for checking language inclusion, based on the observation that $L_{\mathcal{A}} \cup L_{\mathcal{B}} = L_{\mathcal{B}} \Leftrightarrow L_{\mathcal{A}} \subseteq L_{\mathcal{B}}$.

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

$$\begin{aligned} id(R) &= R, \\ r(R) &= \{(X, X) \mid X \subseteq Q\}, \\ s(R) &= \{(Y, X) \mid XRY\}, \\ t(R) &= \{(X, Z) \mid \exists Y \subseteq Q, XRYRZ\}, \\ u(R) &= \{(X_1 \cup X_2, Y_1 \cup Y_2) \mid X_1RY_1 \wedge X_2RY_2\}. \end{aligned}$$

3.2.1 Congruence Algorithm Description

The congruence algorithm works on a similar principle as the antichains algorithm. It starts building \mathcal{A}_{det} and \mathcal{B}_{det} and checks if macrostates in generated pairs are both final or not. The optimization used is based on computing congruence closure of the set of already visited pairs of macrostates. If the generated pair is in this congruence closure, it can be skipped and further not processed. The whole pseudocode of the congruence algorithm is given as algorithm 2.

Algorithm 2: Language equivalence checking with congruence

Input: NFA's $A = (Q_A, \Sigma, \delta_A, s_A, F_A)$, $B = (Q_B, \Sigma, \delta_B, s_B, F_B)$.

Output: TRUE, if $L(A)$ and $L(B)$ are in equivalence relation. Otherwise, FALSE.

```

1 Processed =  $\emptyset$ ;
2 Next =  $\emptyset$ ;
3 insert( $s_A, s_B$ ) into Next;
4 while Next  $\neq \emptyset$  do
5   | extract( $x, y$ ) from Next;
6   | if  $x, y \in c(\textit{Processed} \cup \textit{Next})$  then
7   |   | skip;
8   |   | if  $(x \in F_A \Leftrightarrow y \in F_B)$  then
9   |   |   | return FALSE;
10  |   | insert( $\textit{post}(x, y)$ ) in Next;
11  |   | insert( $x, y$ ) in Processed;
12 return TRUE;
```

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

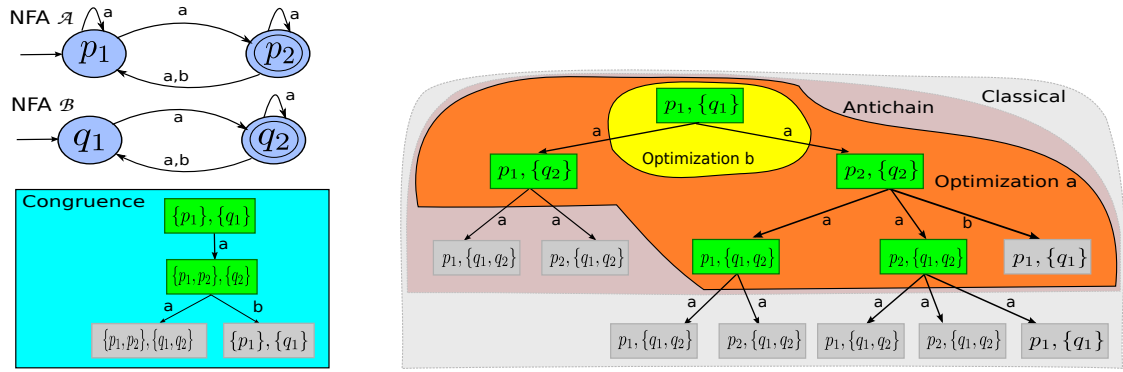


Figure 3.1: The picture is based on an example from [2]. It shows the procedure of checking language inclusion between two NFA using the mentioned approaches (which correspond to the labeled areas). The antichain algorithm reduces number of the generated states compared with the classical, e.g., $(p_2, \{q_1, q_2\})$ is not further explored because $(p_2, \{q_2\}) \sqsubseteq (p_2, \{q_1, q_2\})$. The optimization a and b are improvements of the antichain algorithm using simulation. The congruence algorithm also reduces number of the generated states, so $(\{p_1, p_2\}, \{q_1, q_2\})$ is not further explored because it is in congruence closure of the set of visited states.

Chapter 4

Existing Finite Automata Libraries and the VATA Library

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

As the second VATA library for manipulating of *tree* automata will be introduced. It will be briefly describe library design, operations for tree automata and plans for extension of VATA library.

4.1 Existing Finite Automata Libraries

4.1.1 dk.brics.automaton

dk.brics.automaton is an established Java package available under the BSD license. The latest version of this library (1.11-8) was released on September 7th, 2011. Library can be downloaded and more information are on [13].

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

Test of language inclusion is also supported but if the input automaton is NFA, it needs to be converted to DFA. This is made by *subset construction* approach which is inefficient [6], [2].

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

libfa

libfa is a C library being part of *Augeas* tool. Library is licensed under the LGPL, version 2 or later. It also support both versions of finite automata, NFA and DFA. Regular expressions could serve like input again. *libfa* can be found and downloaded on [12]. *libfa* has no explicit operation for inclusion checking, but has the operations for intersection and complement of automata which can serve for the inclusion checking. Main disadvantage of *libfa* is again need of determinisation.

Fare

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

4.1.2 The RWHT FSA toolkit

The *RWHT FSA* is a toolkit for manipulating finite automata described in [9]. The latest version is 0.9.4 from year 2005. The toolkit is written in C++ and available under its special license, derived from Q Public License v1.0 and the Qt Non-Commercial License v1.0. Library can be downloaded from [10].

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

The RWHT FSA toolkit does not support language inclusion checking explicitly, but contains operations for intersection, complement and determinisation which can be exploited for testing inclusion. This causes that the problem of a state explosion during the explicit determinization of an automaton is still here.

4.1.3 Implementation of New Efficient Algorithms

There have been recently introduced some new efficient algorithms for inclusion checking which are dealing with problem of a state explosion because they avoid the explicit determinization of a finite automaton [6, 2] and [4]. These state-of-the-art algorithms were implemented in OCaml language (mainly) for testing and evaluation purposes.

Some of the mentioned algorithms ([6, ?]) are possible to use not only for finite automata but also for tree automata. These algorithms for tree automata are provided by the VATA library which is implemented in C++ what brings the greater efficiency compared to OCaml implementation. A description of this library will be placed in next section. Despite the fact that a C++ implementation could be more efficient too, there is currently no library or toolkit similar to VATA library providing these algorithms of inclusion checking for finite automata.

4.2 VATA library

4.2.1 General

VATA is a highly efficient open source library for manipulating *non deterministic tree* automata licensed under GPL, version 3. Main application of VATA is in formal verification. VATA library is implemented in C++ and uses the Boost C++ library. Download of library can be found on its website ¹ [11].

Purposes of VATA library are similar as purposes of this work and because VATA also provides basic infrastructure for parsing, serialing and writing finite automata from an input

¹<http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/>

format, it was decided not to create a brand new library, but makes extension of VATA. for finite automata.

4.2.2 Design

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

The main idea of the desing of VATA library is show on the image 4.2.2. Here is also brief description of that idea. The input automata are processed by one of the parsers (currently here is only Timbuk format parser implemented). A result of parsing is a data structure with the information about automaton (it keeps the list of transtion of given automaton, its final states etc.). The main program choose one of the internal encodings of the automata. These encodings contain a definition of a data structure for a representation of automaton and fuctions which create transform the automaton from the data structure created by parser to the data structure of chosen encoding. Each encoding also provide an implementation of the operations over automata. Finally, once all needed operations over the input automata are processed, the result automaton can be dump to a output text format using one of serializers (currently there is only Timbuk format serializer implemented).

As you can see on picture 4.2.2, VATA is written in a modular way, so it is easy to make an extension for finite automata. Thanks to the modularity, any new encoding can share other parts of library such as parser or serializer [11]. VATA also provides a command line interface which is shared by different encodings.

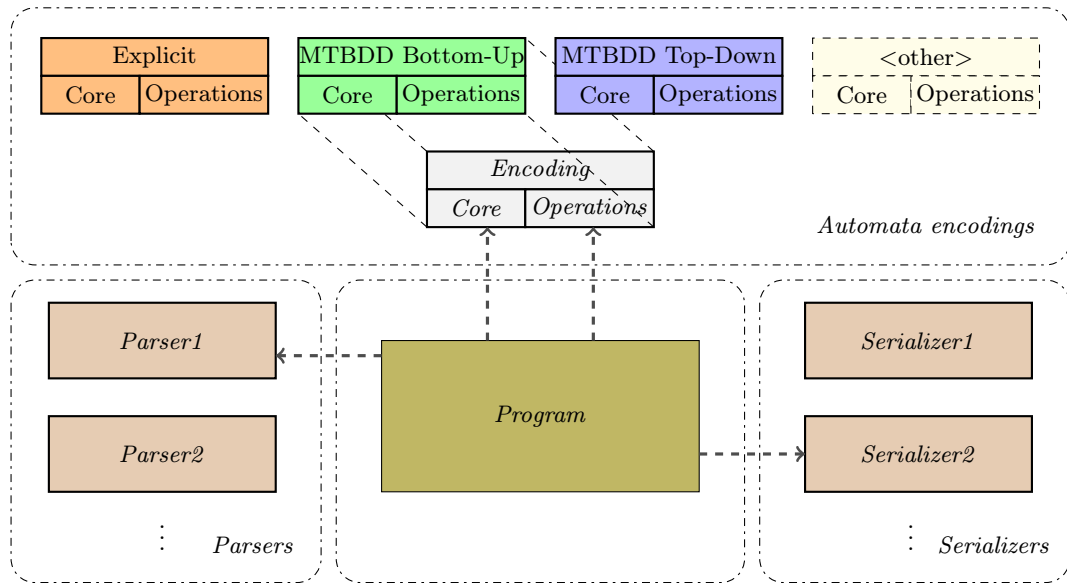


Figure 4.1: The VATA library design. The image is taken from [11]

Explicit Encoding

For storing explicit encoding top-down transitions (transitions are in form $q \xrightarrow{a} (q_1, \dots, q_n)$) is used *hierarchical data structure based on hash tables*. First level of look-up table maps the states to *transition cluster*. This clusters are also look-up table and maps symbols of input alphabet to the set of pointers (stored as *red-black tree*) to tuples of states. Storing tuples of states is of course very memory demanding, so special designed hash table was used for storing them. Inserting new transition to this structure requires a constant number of steps (exception is worst case scenario) [11]. This data structure can be seen on figure 4.2.2.

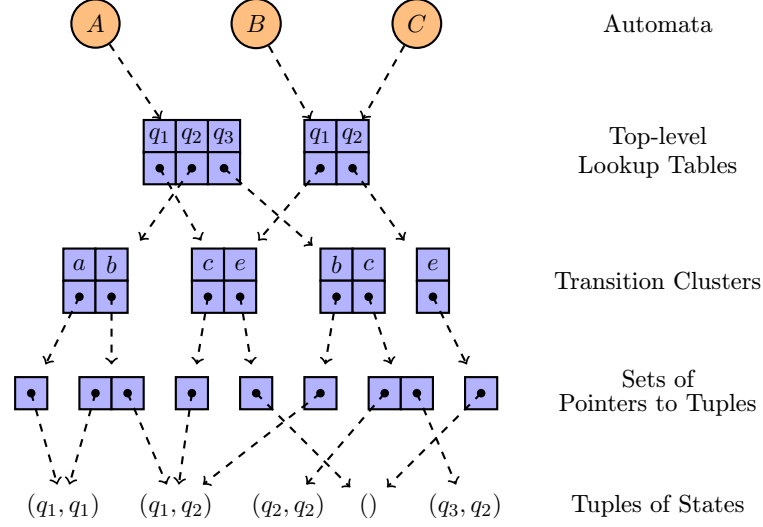


Figure 4.2: The data structure for storing transition of a tree automaton. There is a hash table (top-level lookup table) which maps a state to the pointer to another hash table (transition cluster). Transition cluster maps a symbol of input alphabet to the pointer to the set of pointers to the tuples.

For better performance is used *copy-on-write* technique [11]. The principle of this technique is, that on copy of automaton is created just new pointer to transition table of original automaton and after adding new state to one automaton (original or copy) is modified only part of the shared transition table.

Semi-symbolic Encoding

Transition functions in semi-symbolic encoding are stored in *multi-terminal binary decision diagrams* (MTBDD), which are extension of *binary decision diagrams*. There are provided top-down (transitions are in form $q \xrightarrow{a} (q_1, \dots, q_n)$, for a with arity n) and bottom-up (transitions are in form $(q_1, \dots, q_n) \xrightarrow{a} q$) representation of tree automata in semi-symbolic encoding. The interesting part is saving of symbols in MTBDD. In top-down encoding, the input symbols are stored in MTBDD with their arity, because we need to be able to distinguish between two instances of same symbols with different arity. In opposite case, bottom-up encoding does not need to store arity, because it is possible to get it from arity of tuple on left side of transition [11].

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

Operations

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

For inclusion testing are implemented optimized algorithms from [6, 2]. The inclusion operation is implemented in more versions, so it is possible to use only some heuristic and compare different results.

Efficiency of advanced operations does not come only from the usage of efficient algorithms, but there are also some implementation optimization like *copy-on-write* principle for automata copying (briefly described in subsection 4.2.2), buffering once computed clusters of transitions etc. Other optimization could be found in exploitation of polymorphism using C++ function templates, instead of virtual method because call of virtual function leads to indirect functions call using look-up virtual-method table (because compiler does not know, which function will be called in runtime) what brings an overhead compared to classical direct function call and it also precludes compiler's optimizer to perform some operations [11].

More details about implementation optimization can be found in [11].

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

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

Table 4.1: Table of some supported operations

4.2.3 Extension for Finite Automata

The main goal of this work is to provide operation for language inclusion test of NFA without the need of explicit determinisation. To be precise, VATA library could be already used for finite automata, which can be represented like one dimensional 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 algorithm will be implemented but also

²LTS – Labeled Transitions System

the algorithms for basic operations like such as union, intersection, removing unreachable or useless states etc. This new extension will use the explicit encoding for representing an automaton. The extension will use some already implemented features of VATA like parsing and serializing the input automata or computation of simulation over states of an automaton.

Chapter 5

Design

In this chapter will be described design of the newly created extension of VATA library for finite automata. Firstly, the data structures used for storing a finite automaton will be explain, then principle of translation of the states and the symbols to internal representation and finally chosen input format and its modification.

5.1 Data Structures for Explicit Encoding of Finite Automata

5.1.1 Analysis

An NFA is defined by set of its states, its start and final states (which are subset of all states of a NFA) and also its transitions 2.2.1 and the input alphabet. One needs to keep information about sets of start and final states to be able to distinguish between a start or a final state and the other states. But it is not necessary to store the whole set of states because states that are not start or final are used within transitions. This fact also hold in case of input alphabet.

The set of transitions keeps the most information about an NFA and is also often used during operations over NFA, so the the performance of these operations hardly depend on the efficiency of data structure for a set of transitions. For example, one often wants to get all transitions for given state or for given state and given alphabet symbol. The similiar situaion is in the case of tree automata when it is not necessary to hold the whole set of state but it is important to have efficient data structure for represanting transitions of a given tree automata.

The data structure used for storing transions of a tree automaton in VATA library was described earlier 4.2.2 and can be seen on the figure 4.2.2. The evaluation of VATA library [11] proves efficiency of this data structure is efficient, so it was decided to modificate this data structure and use it also for part of VATA library for finite automata.

5.1.2 Design of Data Structure for Transtions of NFA

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 the transition cluster. The transition cluster is another hash table which maps a symbol of the input alphabet to a set of states. Described data structure is on the figure 5.3.

The data structure for finite automata transitions is simplyfication of the data structure for tree automata what is can be seen by comparing figures 5.3 and 4.2.2. This simplyfication

is possible because the tree version has to store whole tuples of states in transitions. Since these tuples can be very large it is more efficient to store them only once and in data structure keeps pointer to the tuple, which is in a transition, instead of the tuple alone. In case of finite automata this advantage disappears because there are no tuples of state but only states alone and keeping pointer to one state will not bring any memory efficiency (the size of a pointer to a state and the state alone is quite similar). This fact causes that in the data structure for finite automata is not needed to use anything like set of pointers to tuples in tree automata version, but could be directly used the set of states. This set of states would be pointed from transition cluster and would contain all states accessible from a given state under a symbol of input alphabet.

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

The mentioned optimization enables that the tree version of data structure, which has four levels, was simplified to the two level data structure what brings simpler and more efficient manipulation with these data structure.

This data structure also applies the copy-on-write principle, what brings better memory efficiency. It means that the look-up tables and the transition clusters are shared among NFA when they are same and a new look-up table and a transition is created only when a new item is inserted to the one of the automata.

Let give the examples for searching and inserting a transition to this data structure for the NFA on the figure 5.3. If one wants to find all accessible states for state q_1 and symbol a in a NFA A so in the top-level lookup is found pointer to transition cluster for state p . In this transition cluster is symbol a mapped to the set of states (in this case $(\{q_1, q_2\})$) which are accessible from q_1 under a . If one wants to insert a new transition $q_3 \rightarrow eq_2$ to a NFA C , the look-up table pointed by automaton C is duplicated and a state q_3 is inserted to it. A NFA C now points to that newly duplicated look-up table. State q_3 is mapped to pointer to the newly created transition cluster. Symbol e is inserted to this new transition cluster and mapped to the set of state which contains just state q_2 .

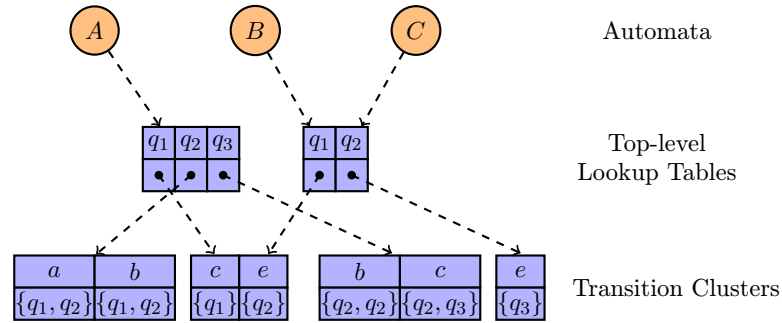


Figure 5.1: The data structure for storing transition of an finite automaton. There is a hash table (top-level lookup table) which map a state of a FA to the pointer to another hash table (transition cluster). Transition cluster maps a symbol of the input alphabet to a set of states.

q_1	q_2	q_3	q_4	a	b	c	d
1	2	3	4	1	2	3	4

Figure 5.2: Translator takes the input format (text format in this case) and maps it to the integer. On the figure can be seen mapping symbols (on the left) and mapping input symbols (on the right).

5.2 Start and final states data structure

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

5.3 Translation of the states and symbols

The input automaton is is parsed and converted to intern representation. During this conversion to the intern representation are states translated (mapped) from input type (e.g., text description of an automaton) to an integers and this also works for symbols of the input alphabet (principle can be seen on figure 5.3). This mechanism brings better efficiency for manipulation with states and symbols during the operations. It also provides unification all input forms to the one internal representation.

The translated integeres can be changed during some operations (e.g., union) over finite automata but the relation between mapping and mapped value will not be broken.

When the input automaton is processed by VATA library operations it is often serialized back to the input format. The result automaton's states are converted back to the input format from internal encoding using translation in reverse order so the result notation stays consistent with the input.

5.4 Usage of the Timbuk format

VATA library provides command-line interface and it is possible to load finite automata from a text file and make some operations over it. The automaton given in a text file needs to have specified format and because there is no standard format for describing the finite automaton, the Timbuk format [1] was chosen as input and output text format of the finite automata. The timbuk format is also used as an format for tree automata in VATA library. In fact, the original purpose of the Timbuk format is describing tree automata but with some modifications can be used also for finite automata. Here is an example of a finite automaton in the Timbuk format:

```
Ops a:1 x:0
Automaton example
States s p q f
Final States f
Transitions
  x → s
  a(s) → p
```

$$\begin{aligned} a(s) &\rightarrow q \\ a(p) &\rightarrow f \\ a(q) &\rightarrow f \end{aligned}$$

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

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

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

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

5.5 Algorithms for Basic Operations

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

5.5.1 Union

The union of two NFA \mathcal{A} and \mathcal{B} is done by following process. First, a brand new automaton is created (this automaton will be result of union). To this automaton are copied sets of start and final states from both original automata. Then the all transitions from \mathcal{A} and \mathcal{B} are added to the newly created automaton. What is the most important during these operations is reindexing of states (it is supposed that the both automata have the same input alphabet so the symbols of it are not reindexed). The reindexing means that there is created index which maps integer that represents a state in original automaton 5.3 to a new integer which will represent the same state in the automaton created by this union.

The reindexing of states is done because the same integer can be used for representing one state of NFA \mathcal{A} and also another state of \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 automaton which is result of the union more readable because the states have the same names as it has in the input automata, only indices 1 and 2 are added for distinguish between states from both automaton are added.

Union of Disjunct States

The special case of union of two NFA is union of disjunct states of these NFA. This is done by copying the one of the NFA to the result automaton and copying of the states (and transitions which contain these states) of the second NFA which are not yet in the result NFA and also copying only of transitions. During this operation no reindexing of states is done.

5.5.2 Intersection

The intersection of two NFA $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$ is defined in preliminaries 2.2.3. In this section we define post-image of the product state $(p, q) \in \mathcal{A} \cap \mathcal{B}$ for a given symbol $a \in \Sigma$ of a NFA $\mathcal{A} \times \mathcal{B}$ by:

$Post_a((p, q)) := \{(p', q') \mid \exists a \in \Sigma : (p, a, p') \in \delta_a, (q, a, q') \in \delta_b\}$. The algorithm for intersection of is described by algorithm 3.

The principle of this algorithm is following. The both NFA are explored parellely and to the result automata are added just product states constisting two states (each from different automaton) that are accessible for given words in the both NFA and to the result automaton are also added only transitions for this states.

Algorithm 3: Algorithm for intersection of NFA

Input: NFA's $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$, $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, \delta_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$
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  $Stack = \emptyset$ ;
2 forall the  $(p_{\mathcal{A}}, p_{\mathcal{B}}) \in I_{\mathcal{A}} \times I_{\mathcal{B}}$  do
3   Add  $(p_{\mathcal{A}}, p_{\mathcal{B}})$  to  $I_{\mathcal{A} \cap \mathcal{B}}$ ;
4   Push  $(p_{\mathcal{A}}, p_{\mathcal{B}})$  on  $Stack$ ;
5   if  $(p_{\mathcal{A}} \in F_{\mathcal{A}} \wedge p_{\mathcal{B}} \in F_{\mathcal{B}})$  then
6     Add  $(p_{\mathcal{A}}, p_{\mathcal{B}})$  to  $F_{\mathcal{A} \cap \mathcal{B}}$ 
7 while  $(Stack \neq \emptyset)$  do
8   Pick and remove a product-state  $(p_{\mathcal{A}}, p_{\mathcal{B}})$  from  $Stack$ ;
9   forall the  $(q_{\mathcal{A}}, q_{\mathcal{B}}) \in Post_a(p_{\mathcal{A}}, p_{\mathcal{B}})$  do
10    if  $(q_{\mathcal{A}} \in F_{\mathcal{A}} \wedge q_{\mathcal{B}} \in F_{\mathcal{B}})$  then
11      Add  $(q_{\mathcal{A}}, q_{\mathcal{B}})$  to  $F_{\mathcal{A} \cap \mathcal{B}}$ 
12    Add  $(p_{\mathcal{A}}, p_{\mathcal{B}}) \xrightarrow{a} (q_{\mathcal{A}}, q_{\mathcal{B}})$  to  $\delta_{\mathcal{A} \cap \mathcal{B}}$ ;
13    Push  $(q_{\mathcal{A}}, q_{\mathcal{B}})$  on  $Stack$ ;
14 return NFA  $\mathcal{A} \cap \mathcal{B}$ ;

```

5.5.3 Reverse

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

Algorithm 4: Algorithm for reverting of an NFA

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

```

1  $F_{\mathcal{A}_{rev}} = I_{\mathcal{A}}$ ;
2  $I_{\mathcal{A}_{rev}} = F_{\mathcal{A}}$ ;
3 forall the  $(p, a, q) \in \delta_{\mathcal{A}}$  do
4   Add  $(q, a, p)$  to  $\delta_{\mathcal{A}_{rev}}$ ;
5 return NFA  $\mathcal{A}_{rev}$ ;

```

5.5.4 Removing Unreachable States

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

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

Algorithm 5: Algorithm for removing the unreachable states of NFA

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

```

1 Reachable =  $I_{\mathcal{A}}$ ;
2 Stack = Reachable;
3 while (Stack  $\neq \emptyset$ ) do
4   Pick and remove a  $p$  from Stack;
5   forall the  $q \in \{q' \mid \exists a \in \Sigma : (p, a, q') \in \delta_{\mathcal{A}}\}$  do
6     if ( $q \notin \textit{Reachable}$ ) then
7       Push  $q$  on Stack;
8       Add  $q$  to Reachable;
9  $I_{\mathcal{B}} = I_{\mathcal{A}}$ ;
10 forall the  $p \in \textit{Reachable}$  do
11   if  $p \in F_{\mathcal{A}}$  then
12     Add  $p$  to  $F_{\mathcal{B}}$ ;
13   Add  $\{(p, a, q) \mid \exists a \in \Sigma \wedge q \in Q_{\mathcal{A}} : (p, a, q) \in \delta_{\mathcal{A}}\}$  to  $\delta_{\mathcal{B}}$ ;
14 return NFA  $\mathcal{B}$ ;
```

5.5.5 Removing Useless States

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

5.5.6 Get Candidate

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

The operation for getting candidate is implemented by the algorithm 6. This algorithm copies the set of start states of \mathcal{A} to the set of start states of \mathcal{B} and also add this set to a set of reachable states. Then all transitions for states in a reachable state's set that are

in $\delta_{\mathcal{A}}$ are added to \mathcal{B} and finally states accessible from current set of reachable states are added to this set. This is repeated until the whole NFA \mathcal{A} is copied to the NFA \mathcal{B} or the final state is not in set of the newly added states.

Algorithm 6: Algorithm for getting witness in NFA

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

```

1  $I_{\mathcal{B}} = I_{\mathcal{A}};$ 
2  $Reachable = I_{\mathcal{A}};$ 
3  $Stack = Reachable;$ 
4 while ( $Stack \neq \emptyset$ ) do
5   Pick and remove a  $p$  from  $Stack$ ;
6   forall the  $\{(p, a, q) \mid \exists a \in \Sigma : (p, a, q) \in \delta_{\mathcal{A}}\}$  do
7     if ( $q \notin Reachable$ ) then
8       Push  $q$  on  $Stack$ ;
9       Add  $q$  to  $Reachable$ ;
10    Insert  $(p, a, q)$  to  $\delta_{\mathcal{B}}$ ;
11    if  $q \in F_{\mathcal{A}}$  then
12      Add  $q$  to  $F_{\mathcal{B}}$ ;
13      return NFA  $\mathcal{B}$ ;
14 return NFA  $\mathcal{B}$ ;
```

Chapter 6

Implementation

This chapter provides description of the module of VATA library for the finite automata. The description of the loading of an finite automata will be given at first. Then the operations for basic operations like union or intersection will be explained and finally the algorithms (based on bisimulation up-to congruence and on the antichain is given) for testing inclusion of the finite automata is given.

6.1 Loading and Manipulation with an Finite Automata in the Explicit Encoding

Loading of the finite to the explicit is done by class *ExplicitFiniteAut* what is the main class for representation of an finite automaton. This class has the data members that implements the data structure for explicit encoding of an finite automaton described in previous chapter 5. It also provides operations for manipulation with an automaton like setting specific state as start or final one. The class *ExplicitFiniteAut* also ensures translation 5.3 of the states and symbols to the internal representation of integers.

6.2 Used parts of existing implementation of VATA library

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

Parser and Serializer

For loading automaton from text specification is used module of VATA library called *parser* and for serializing back to this specification module called *serializer*. Because for finite automata have been used the same input and output text format ?? it is possible to use the original parse and serializer for this format which have been implemented in the existing part of library. The parser returns a data structure which generally describes a finite automaton and this data structure is further processed in part of library for finite automata where is converted to the data structure for explicit encoding of the finite automata. When one wants to dump an automaton from explicit encoding back to the text format, the automaton is converted to a data structure which is identical with a data structure returned by parser.

Description of an automaton in this data structure is given to the serializer which dumps it to the text format.

Simulation

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

Utilities

The original VATA library also provides lot of utilities which are also usefull for implementation of extension for the finite automata. These utilities provides classes for easier processing of finite automata. For example, the classes *TwoWayDict* and *TranslatorStrict* are uses for conversion of an finite automaton to the explicit encoding, the class *Antichain2Cv2* is used for storing states to the antichain during checking inclusion using antichain algorithm 3.1 and the class *AutDescription* for represanting an automaton after the parsing.

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

6.3 implementation of antichain algorithms

6.4 improvments for simulation

6.5 Implementation of Congruence Closure

The algorithm using bisimilation up to congruence for checking inclusion of languages of NFA is described in its own section (3.2). This algorithm reduces number of macro states that are neccessary to check to determine if inclusion holds or not thanks to the computation of a congruence closure of a relation of the visited macro states because if the newly generated macro state is in the congruence closure it is not neccessary to explore macro state which are accessible from this macro state.

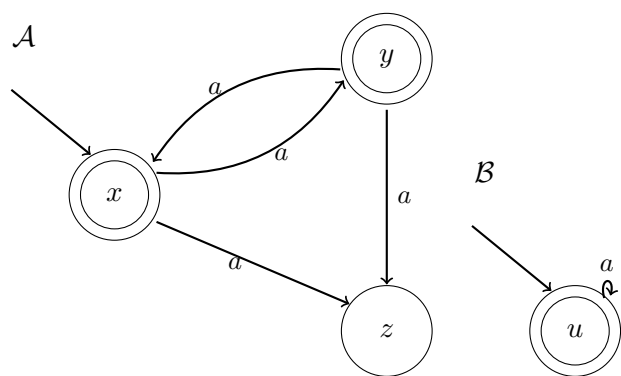
The implementation of the computation of the congruence closure is crucial for performance and efficiency of the whole method. In VATA library is used an algorithm described by [4] which is based on using of the so-called rewriting rules. For each pair of macro states (X, Y) in a relation R of the visited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

$$X \rightarrow X + Y \qquad Y \rightarrow X + Y$$

These rules can be used for computing a *normal form* of a set of states [4]. The normal form of a set of states X created with usage of rewriting rules of a relation R is denoted as $X \downarrow_R$.

One can check if $(X, Y) \in c(R)$ using the principle of rewriting rules because once there are computed normal forms $X \downarrow_R$ and $Y \downarrow_R$ so $X \downarrow_R = Y \downarrow_R$ holds iff $(X, Y) \in c(R)$ [4].

An example (taken from [4]) of applying this approach for checking inclusion of NFA \mathcal{A} and \mathcal{B} (both are on figure ??) is given on the figure ??.



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

Chapter 7

Experimental evaluation

7.1 improvemnet give by congrunce

Chapter 8

Conclusion

8.1 futher development

Bibliography

- [1] Timbuk. <http://www.irisa.fr/celtique/genet/timbuk/>, 2012 [cit. 2013-01-29].
- [2] Abdulla, Parosh Aziz and Chen, Yu-Fang and Holík, Lukáš and Mayr, Richard and Vojnar, Tomáš. When simulation meets antichains: on checking language inclusion of nondeterministic finite (tree) automata. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 158–174, Berlin, Heidelberg, 2010. Springer-Verlag.
- [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 *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 457–468, New York, NY, USA, 2013. ACM.
- [5] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract Regular Model Checking. In *Computer Aided Verification*, Lecture Notes in Computer Science, pages 372–386. Springer Verlag, 1995.
- [6] De Wulf, M. and Doyen, L. and Henzinger, T. A. and Raskin, J. -F. Antichains: a new algorithm for checking universality of finite automata. In *Proceedings of the 18th international conference on Computer Aided Verification*, CAV'06, pages 17–30, Berlin, Heidelberg, 2006. Springer-Verlag.
- [7] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation (PLDI'02)*, pages 69–82. ACM, 2002.
- [8] 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 *In practice, in tools and algorithms for the construction and analysis of systems, first international workshop (TACAS '95)*. Springer Verlag, 1995.
- [9] 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.
- [10] Stephan Kanthak and Hermann Ney. The RWTH FSA Toolkit. <http://www-i6.informatik.rwth-aachen.de/~kanthak/fsa.html>, 2005 [cit. 2013-01-19].

- [11] Lengál, Ondřej and Šimáček, Jiří and Vojnar, Tomáš. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In *Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 79–94, Berlin, Heidelberg, 2012. Springer-Verlag.
- [12] David Lutterkort. libfa. <http://augeas.net/libfa/index.html>, 2011 [cit. 2013-01-19].
- [13] Anders Møller. dk.brics.automaton. <http://www.brics.dk/automaton/>, 2011 [cit. 2013-01-19].

Appendix A

Storage Medium