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 MARTIN HRUŠKA
AUTHOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EFEKTIVNÍ ALGORITMY PRO PRÁCI S KONEČNÝMI AUTOMATY

EFFICIENT ALGORITHMS FOR FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN HRUŠKA
AUTHOR

VEDOUCÍ PRÁCE Ing. ONDŘEJ LENGÁL
SUPERVISOR

BRNO 2013

Abstrakt
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 18, 2013

Poděkování

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

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 **19**
5.1 Data Structure for Explicit Encoding of Finite Automata 19
5.2 Analysis 19
5.2.1 Symplifying of the original data structure 19
5.2.2 Keeping the start states 19
5.2.3 Translation of the states and symbols 19
5.3 Overview of used interfaces of VATA library 19
5.4 Usage of timubk format 19

6 Implementation **20**
6.1 implementation of basic operations 20
6.2 implementation of antichain algorithms 20
6.3 improvments for simulation 20
6.4 implementation of congruence closure 20

7 Experimental evaluation **21**
7.1 improvemnet give by congrunce 21

8 Conclusion **22**
8.1 futher development 22

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 [4], 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 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 [5].

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* [5, 1] or the so-called *bisimulation up to congruence* [3]. 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 [10], 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 5. 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 fundations 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 .

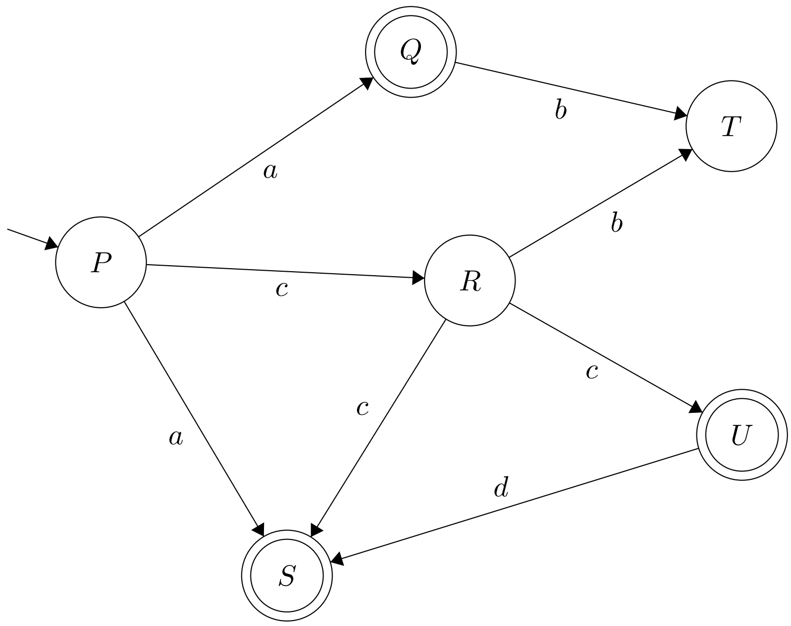


Figure 2.1: An example of a NFA

2.2.2 Deterministic Finite Automaton

A *deterministic finite automaton* (DFA) is a special case of an NFA, where δ is a partial function $\delta : Q \times \Sigma \to Q$ 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, 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

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

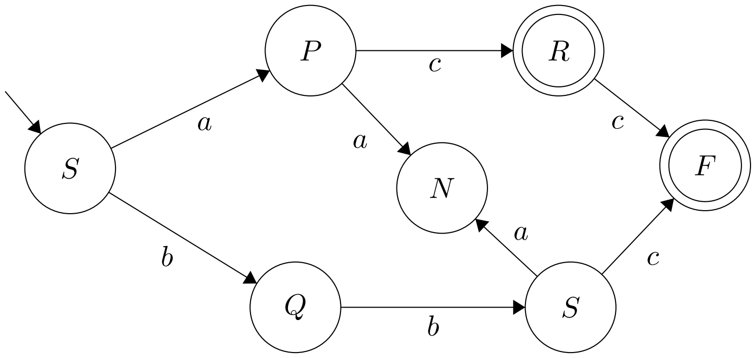


Figure 2.2: An example of a DFA

Automata Intersection

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

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

where δ is defined by

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

Automata Product

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

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

where δ is defined by

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

Subset construction

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

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

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

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

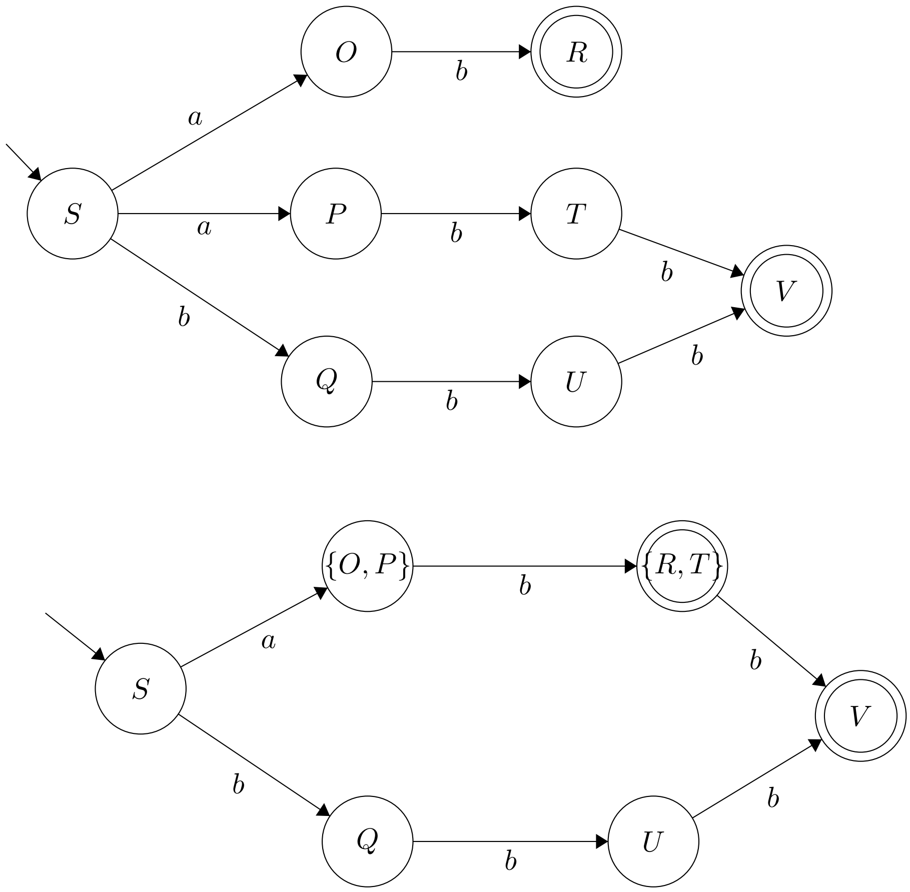


Figure 2.3: A simple example of NFA to DFA conversion via the subset construction. Here is shown small NFA with small Σ, but for larger NFA could state explosion occur.

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 $q_n \in F$. An word $w \in \Sigma^*$ is called *accepting*, if there exists an *accepting* run for w. An *unreachable* state q of an NFA $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ 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 $r = q \dots q$ of A over a word $w \in \Sigma^*$ such that $q_n \in F$. Given a pair of states p, q of an NFA $A = (Q, \Sigma, \delta, I, F)$, these states are equivalent if $\forall w \in \Sigma^* :$ *Run from p over w is accepting* $\Leftrightarrow$ *Run from q over w is accepting.*

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

2.3 Regular Languages

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

2.3.1 Closure Properties

Regular languages are closed under certain operation, if result of this operation 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 $L_\mathcal{A} \cap \overline{L_\mathcal{B}} = \varnothing$. This problem is PSPACE-complete [5]. The *textbook* algorithm for checking inclusion $L_\mathcal{A} \subseteq L_\mathcal{B}$ works by first determinizing $\mathcal{B}$ (yielding the DFA $\mathcal{B}_{det}$ using subset construction algorithm 2.2.3), complementing it ($\overline{\mathcal{B}_{det}}$) and constructing the NFA $\mathcal{A} \times \overline{\mathcal{B}_{det}}$ accepting the intersection of $L_\mathcal{A}$ and $L_{\overline{\mathcal{B}_{det}}}$ and checking whether its language is nonempty. Any accepting run in this automaton may serve as a witness that the inclusion between $\mathcal{A}$ and $\mathcal{B}$ does not hold. Some recently introduced approaches (so-called antichains [5], its optimization using simulation [1] and so-called bisimulation up to congruence [3]) 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 futher 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 $A \times B$ 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 A and P is rejecting in automaton B.

3.1.1 Antichain Algorithm Description

The antichains algorithm [5] 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 $r = p \land R \subseteq P$. If there is generated a pair (p, P) and there is $(r, R) \in X$ such that $(r, R) \sqsubseteq (p, P)$, we can skip (p, P) and not insert it to X for further search.

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

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

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

Algorithm 1: Language inclusion checking with antichains and simulations

Input: NFA's $\mathcal{A} = (Q_A, \Sigma, \delta_A, S_A, F_A)$, $\mathcal{B} = (Q_B, \Sigma, \delta_B, S_B, F_B)$.
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* $\{(s, S_{\mathcal{B}}) | s \in S_{\mathcal{A}}\}$ **then**
2   return FALSE;
3 Processed:=$\emptyset$;
4 Next:= Initialize($\{(s, Minimize(S_B)) \mid s \in S_A\}$);
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** $(p, P) \in \{(r', Minimize(R')) \mid (r', R') \in Post((r, R))\}$ **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** $\nexists (x, X) \in Processed \cup Next$ *s.t.* $p \preceq x \land X \preceq^{\forall\exists} P$ **then**
13   Remove all (x, X) from $Processed \cup Next$ *s.t.* $x \preceq p \land P \preceq^{\forall\exists} X$;
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[3]. The definition of congruence relation is following:

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

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

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

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

$$id(R) = R,$$

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

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

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

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

3.2.1 Congruence Algorithm Description

The congruence algorithm works on a similar principle as the antichains algorithm. 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.

$Processed = \varnothing$;
$Next = \varnothing$;
insert(s_A,s_B) into $Next$;
while $Next \neq \varnothing$ **do**
  extract(x,y) from $Next$;
  if x,$y \in c(Processed \cup Next)$ **then**
    skip;
  if ($x \in F_A \Leftrightarrow y \in F_B$) **then**
    return FALSE;
  insert($post(x, y)$) in $Next$;
  insert(x,y) in $Processed$;
return TRUE;

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

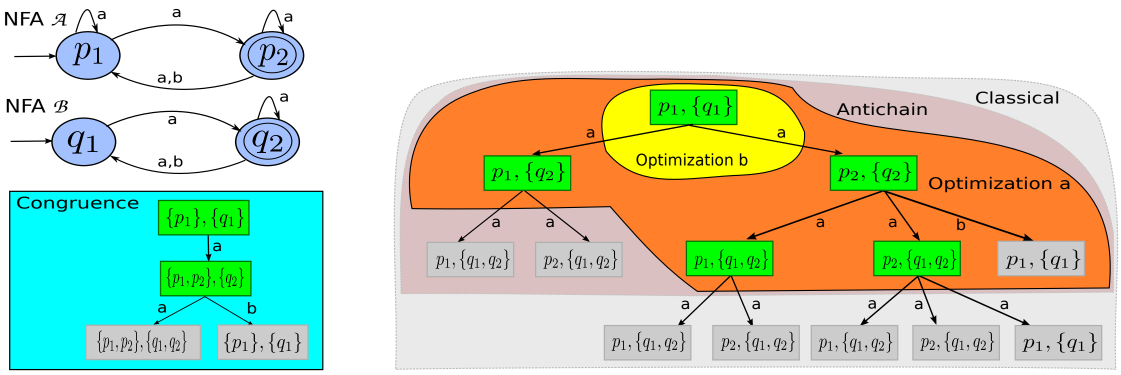


Figure 3.1: The picture is based on an example from [1]. It shows the procedure of checking language inclusion between two NFA using the mentioned approaches (which correspond to the labeled areas). The antichain algorithm reduces number of the generated states compared with the classical, e.g., $(p_2,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

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 [12].

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 [5], [1].

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 [11]. 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 [2].

4.1.2 The RWHT FSA toolkit

The *RWHT FSA* is a toolkit for manipulating finite automata described in [8]. 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 [9].

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 immediatelly 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 [5, 1] and [3]. These state-of-the-art algorithms were implemented in OCaml languaue (mainly) for testing and evaluation purposes.

Some of the mentioned algorithms ([5, **?**]) are possible to use not only for finite automat 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 implmentation. 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 similiar 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] [10].

Purposes of VATA library are similar as purposes of this work and becasue 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/`

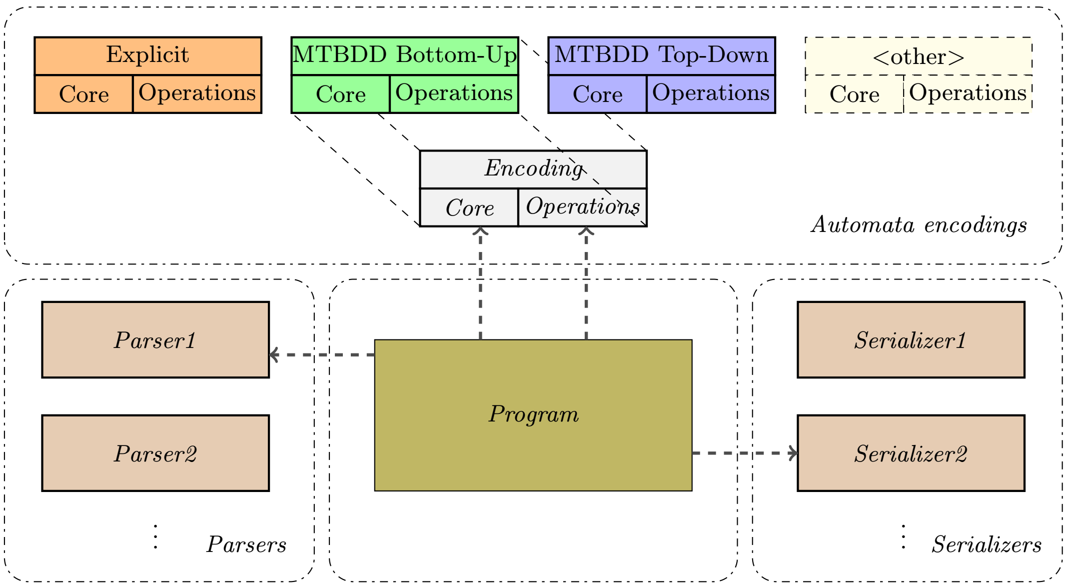
format, it was decided not to create a brand new library, but makes extension of VATA. for finite automata.

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 progam choose one of the internal encodings of the automata. These encodings contain a definiton 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 [10]. VATA also provides a command line interface which is shared by different encodings.



Explicit Encoding

For storing explicit encoding top-down transitions (transitions are in form $q \xrightarrow{a} (q_1, ..., q_n)$) 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) [10].

For better performance is used *copy-on-write* technique [10]. 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 $q \xrightarrow{a} (q_1, ..., q_n)$, for a with arity n) and bottom-up (transitions are in form $(q_1, ..., q_n) \xrightarrow{a} q$) representation of tree automata in semi-symbolic encoding. Interesting 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 [10].

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 [5, 1]. 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 comapred to classical direct function call and it also precludes compiler's optimizer to perform some operations [10].

More details about implementation optimization can be found in [10].

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

[2] LTS – Labeled Transitions System

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 the algoritms 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. Firstly, data structures, which have been used for representation of finite automata in the explicit encoding, will be described. These data structures was created by simplifying the data structure for representing tree automata in explicit encoding so the comparsion of both will be given.

5.1 Data Structure for Explicit Encoding of Finite Automata

5.2 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. But not all of the parts of the definition have to be saved alone. The set of states will be implicitly given by transitions and there is no need to save them alone in some special set. The same fact holds for input alphabet. On the other hand, it is neccessesary to create special data structure for start state and final state to be able to distinguish between start state or final state and ordinary state of an NFA, what is not possible to recoqnize from transtion alone.

While a data structure for storing the final and start states can be in set, data structure for representing the transtions of an NFA is more complicated.

For the explicit encoding of tree automata is used data structure described earlier in this documents 4.2.2. The evaluation of performance of VATA library was proven that this data structure works quite well.

5.2.1 Symplifying of the original data structure

5.2.2 Keeping the start states

5.2.3 Translation of the states and symbols

5.3 Overview of used interfaces of VATA library

5.4 Usage of timubk format

Chapter 6

Implementation

6.1 implementation of basic operations

6.2 implementation of antichain algorithms

6.3 improvments for simulation

6.4 implementation of congruence closure

Chapter 7

Experimental evaluation

7.1 improvemnet give by congrunce

Chapter 8

Conclusion

8.1 futher development

Bibliography

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[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 *In practice, in tools and algorithms for the construction and analysis of systems, first international workshop (TACAS '95)*. Springer Verlag, 1995.

[8] 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.

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

[10] 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.

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

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