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
March 29, 2013

Poděkování

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

Contents

Chapter 1

Introduction

A finite automaton (FA) is a model of computation with applications in different branches of computer science, e.g., compiler design, formal verification, 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 [12], 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

Finite Automata and Languages

This chapter contains theoretical fundations of the thesis. No proofs are given, because they can be found in literature. First, the finite automaton and their context will be defined, then the language, regular language and its closure properties and at last conversion from NFA to DFA with subset construction.

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 $w = a_1 \ldots a_n$, 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 $u = a_1 \ldots a_2$ and $v = b_1 \ldots b_n$ over Σ it holds that $\epsilon \cdot u = u \cdot \epsilon = u$ and $u \cdot v = a_1 \ldots a_n b_1 \ldots b_m$. 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 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* and *positive* iteration 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 $N = (Q, \Sigma, \delta, I, F)$, where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation. 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 .

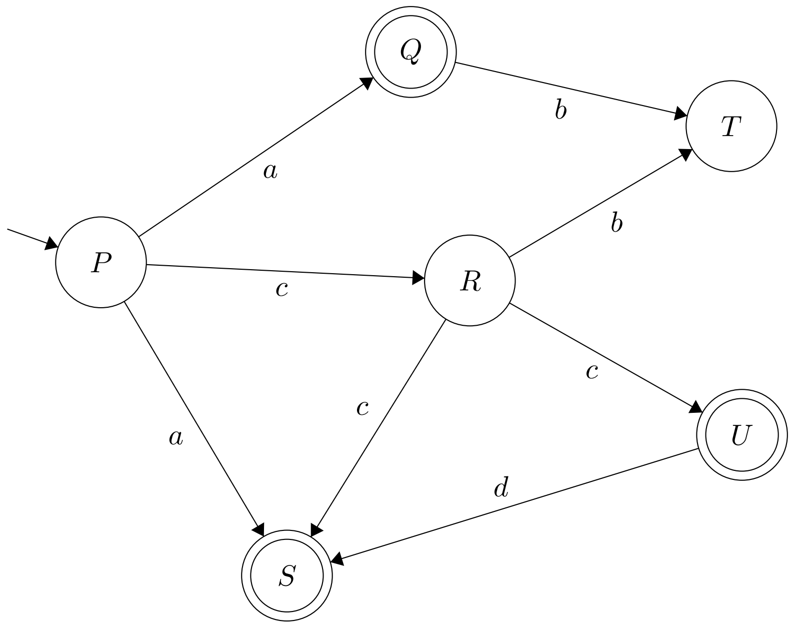

Figure 2.1: An example of a NFA

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 $N = (Q, \Sigma, \delta, I, F)$ 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

Definition 2.2.1. *$A = (Q_A, \Sigma, \delta_A, I_A, F_A)$ and $B = (Q_B, \Sigma, \delta_B, I_B, F_B)$ are two NFA's. Their union is defined by*

$A \cup B = (Q_A \cup Q_B, \Sigma, \delta_A \cup \delta_B, I_A \cup I_B, F_A \cup F_B)$

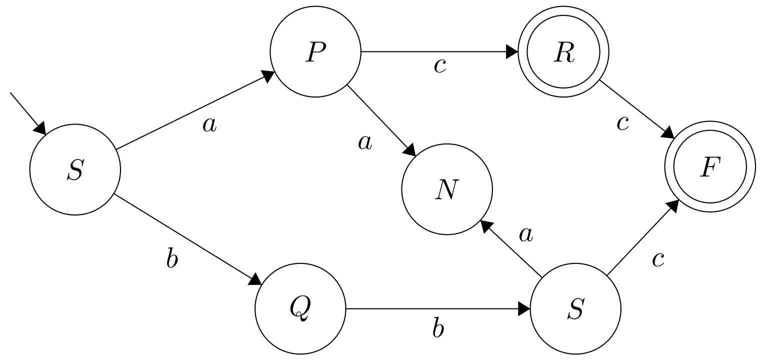

Figure 2.2: An example of a DFA

Automata Intersection

Definition 2.2.2. *$A = (Q_A, \Sigma, \delta_A, I_A, F_A)$ and $B = (Q_B, \Sigma, \delta_B, I_B, F_B)$ are to two NFA's. 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

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

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 $w = a_1 \ldots a_n$ is a sequence $r = q_0 \ldots q_n$, 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 ω. An *unreachable* state q of an NFA $A = (Q, \Sigma, \delta, I, F)$ is a state for which there is no run $r = q_0 \ldots q$ of 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 $r = q \ldots q$ 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 $\forall \omega \in \Sigma^* :$ *Run from p over ω is accepting* $\Leftrightarrow$ *Run from q over ω is accepting.*

2.2.5 Minimum DFA

Definition 2.2.3. *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 $L_{\mathcal{A}}(q) = \{w \in \Sigma^* \mid$ *there exists an accepting run of $\mathcal{A}$ from q over w*$\}$, while the language of a set of states $R \subseteq Q$ is defined as $L_{\mathcal{A}}(R) = \bigcup_{q \in R} L_{\mathcal{A}}(q)$. The language of an NFA $\mathcal{A}$ is defined as $L_{\mathcal{A}} = L_{\mathcal{A}}(I)$.

2.3 Languages

2.3.1 Language Accepted by Finite Automaton M

Definition 2.3.1. The language accepted by finite automaton M *is the set of all strings accepted by* M *and is denoted as* $L(M)$*:*

$$L(M) = \{\omega \in \Sigma^* \mid \text{there exists an accepting run for } \omega \text{ in } L\}$$

Definition 2.3.2. $L(M)(p)$ *is language of state* p *from finite automaton* M*.*

$$L(M)(p) = \{\omega \in \Sigma^* \mid \text{there exists an accepting run of } M \text{ over } \omega \text{ in } L, \text{ which starts in state } p\}$$

2.3.2 Regular Language

Definition 2.3.3. *A language* L *is* regular*, if exists some finite automaton* M*, that* $L = L(M)$*.*

2.3.3 Closure Properties of Regular Languages

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 alphabet Σ:

- Union: $L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$
- Intersection: $L_1 \cap L_2 = \{x \mid x \in L_1 \wedge x \in L_2\}$
- Complement: $\overline{L} = \{x \mid x \notin L\}$
- Difference: $L - K = \{x \mid x \in L \wedge x \notin K\}$
- Reversal: $L^R = \{y \mid x = a_1 \dots a_n \in L \Rightarrow y = a_n \dots a_1\}$
- Kleen closure (star): L^* is defined as:
  - $L_0 = \{\epsilon\}$, where epsilon is empty string
  - $L_1 = \{a\}$, where $a \in \Sigma$
  - $L_{i+1} = \{wv \mid w \in L_i \wedge v \in \Sigma\}$, where $i > 0$
- Concatenation: $LK = \{xy \mid x \in L \wedge y \in K\}$
- Homomorphism: h is homomorphism on alphabet Σ of language L.

  $h(L) = \{h(x) \mid x \in L\}$
- Inverse homomorphism: h is homomorphism on alphabet Σ of language L.

  $h^{-1}(L) = \{x \mid h(x) \in L\}$

2.3.4 Language inclusion

Definition 2.3.4. *Let have language of finite automaton* A *and language of finite automata* B*. Inclusion of this two languages is defined as:*

$$L(A) \cap \overline{L(B)} = \varnothing$$

2.4 Subset construction

Now we will define how to construct equivalent DFA A_{det} for a given NFA $A = (Q, \Sigma, \delta, S, F)$. This classical („textbook") approach is called *subset construction*.

Definition 2.4.1. $A_{det} = (2^Q, \Sigma, \delta_{det}, S, F_{det})$, *where*

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

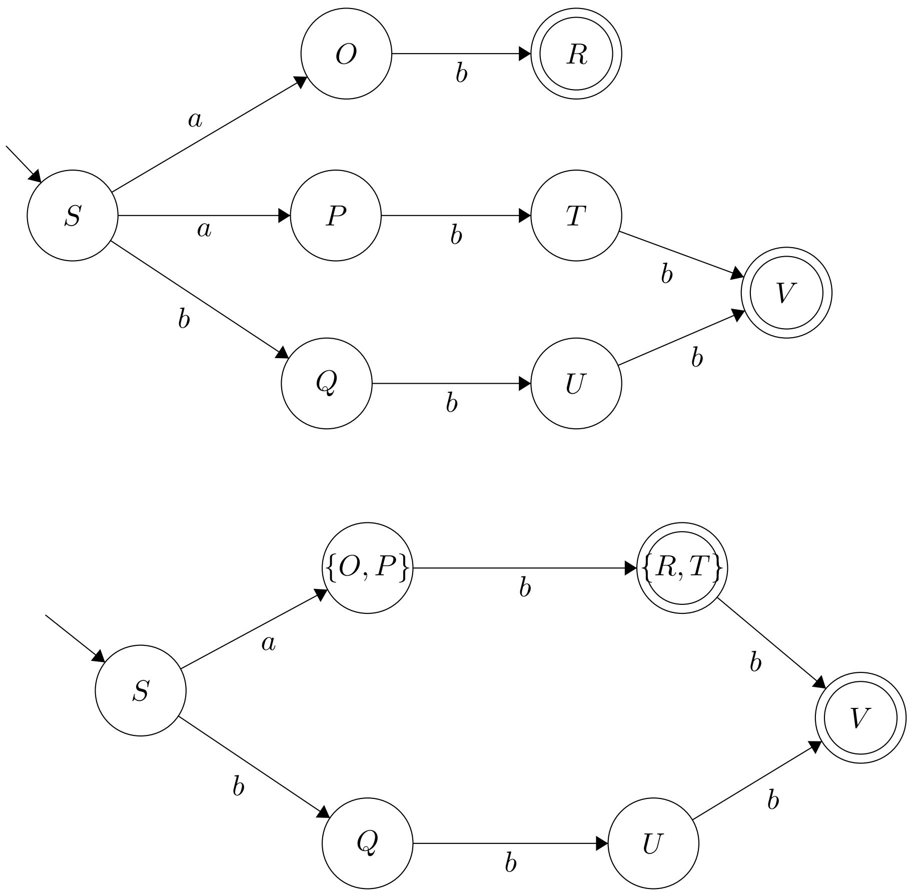

Figure 2.3: Simple example of NFA to DFA conversion via subset construction. Here is small NFA with small Σ, but for larger NFA could state explosion occur.

Chapter 3

Inclusion Checking of NFA

Language inclusion problem is decision whether $L(A) \subseteq L(B)$. This problem is PSPACE-complete [6]. For decision of *language inclusion problem* exists some classical algorithms. Classical algorithms are based on searching of accepting state of product automaton $A \times \overline{B}$. Such state can serve like *witness*, that there is no inclusion between A and B. Problem of classical (textbook) algorithm is that they are based on explicit determinisation of automaton B via *subset construction.* Of course, if *witness* state is found, algorithm ends, but very often whole DFA is build, what leads to state explosion. But many of states are not in fact necessary for inclusion testing.

In this work are used two approaches, which do not need explicit determinisation of NFA, first uses *antichains and simulations* [6, 2] for pruning states of NFA and second uses *bisimulation up to congruence* [4].

Let denote product state as pair (p, P), where $p \in A \wedge P \in B$

For these algorithms we have to define post-image of product state of automaton $A \times B$:

Definition 3.0.2. *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

First approach uses optimized method based on combination of *antichains with simulation* from [2]. Firstly, let define the antichain and simulation ([9]).

Definition 3.1.1. *Let S be a* partially order *set. Two elements $a, b \in S$ are incomparable iff neither $a \leq b$, nor $b \leq a$. An* antichain *is a set $A \subseteq S$, where*

$$A \subseteq S = \{\forall a, b \in A \ . \ a \notin b \cap b \not\ni a\}.$$

Definition 3.1.2. *A forward* simulation *on NFA $A = (Q, \Sigma, \delta, I, F)$ is a relation $\preceq \subseteq Q \times Q$ such that $p \preceq r$ iff (i) $p \in F \Rightarrow r \in F$ and (ii) for every transition $p \xrightarrow{a} p'$, there exists a transition $r \xrightarrow{a} r'$ such that $p' \preceq r'$*

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

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

Pseudocode of *antichain* approach is 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:=$\varnothing$;
4 Next:= Initialize($\{(s, Minimize(S_B)) \mid s \in S_A\}$);
5 **while** *(Next $\neq \varnothing$)* **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.* $p \preceq x \wedge X \preceq^{\forall\exists} P$ **then**
13   Remove all (x, X) from $Processed \cup Next$ *s.t.* $x \preceq p \wedge P \preceq^{\forall\exists} X$;
14   Add (p, P) to Next;
15 **return** TRUE;

Let describe two optimization use in algorithm 1 based on [2]. First optimization comes from the observation that we can stop search from product-state (p, P), if there exists some visited product state (r, R), such that $p \preceq r \wedge R \preceq^{\forall\exists} P$, or $\exists p' \in P : p \preceq p'$. First part of condition says that if (p, P) takes automaton to the accepting state, (r, R) will be taken to accepting state too, so we do not have search from (p, P). Second part of condition shows, that every accepting word of p takes P to accepting state too, because $\exists p' \in P : p \preceq p'$. Proof for this can be found in [2]. This first optimization is in algorithm 1 at lines 11–14.

Second optimization is based on fact, that $L(A)(P) = L(A)(P - \{p_1\})$, if there exists p_2, such as $p_1 \preceq p_2$. We can remove the state p_1 from macro-state P, because if $L(A)(P)$ rejects the word, then $L(A)(P - \{p_1\})$ rejects this word too. This optimization is applied in function $Minimize$ at lines 4 and 7 in algorithm 1. Proof for this optimization is again in [2].

3.2 Checking Inclusion with Bisimulation up to Congruence

Checking inclusion with using *bisimulation up to congruence* is based on *Hopcroft and Karp's* algorithm. Congruence algorithm serves for checking language equivalence, but it can be used also for checking language inclusion. Indeed, let X and Y be sets of states of NFA's. So $X \cup Y = Y$, iff $X \subseteq Y$. It is possible to check equivalence for $X + Y$ and Y [4].

Before introduction of the algorithm itself, we will define *congruence* relation.

Definition 3.2.1. *Let X be a set with n-ary operation O over X.* Congruence *is an equivalence relation R, which follows this condition* $\forall a_1, \ldots, a_n, b_1, \ldots, b_n \in X$*:*

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

where $a_i \in X, b_i \in X$

Optimized algorithm uses *congruence closure function* c for pruning out the unnecessary states for checking equivalence. Pseudocode is 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 = ∅;
Next = ∅;
insert(s_A,s_B) into Next;
while Next ≠ ∅ do
    extract(x,y) from Next;
    if x,y ∈ c(Processed ∪ Next) then
        skip;
    if (x ∈ F_A ⇔ y ∈ F_B) then
        return FALSE;
    insert(post(x, y)) in Next;
    insert(x,y) in Processed;
return TRUE;
```

Chapter 4

Existing Finite Automata Libraries and VATA library

There are many different libraries for finite automata. Libraries have 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 established Java package available under BSD license. Last version of this library (1.11-8) was released on September 7th, 2011. Library can be downloaded and more information are on [14].

Library can use as input regular expression created by Java class *RegeExp*. 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 implemented in C. *libfa* is 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 [13]. *libfa* has no explicit operation for inclusion checking, but has operations for intersect and complement of automata, which can serve for inclusion checking. Main disadvantage of libfa is again need of determinisation.

Fare

Fare is library, which brings *dk.brics.automaton* from Java to .NET. This library has same characteristics like *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 toolkit for manipulating finite automata described in [10]. The latest version is 0.9.4 from year 2005. Toolkit is written in C++ and available under its special license, derived from Q Public License v1.0 and the Qt Non-Commercial License v1.0. Library can be downloaded from [11].

Library supports on-demand computation for better memory efficiency. Another advantage is, that it supports the weighted transitions, so it is possible to use it for wider range of applications. The RWHT FSA toolkit does not support the explicitly directly, but contains operations for intersection, complement and determinisation, which are unnecessary for inclusion testing. There are used some optimization, but this not compensates need of determinisation.

4.1.3 Implementation of New Efficient Algorithms

New efficient algorithms for inclusion testing, which were introduced in [6, 2] and [4], was implemented for finite automata only in OCaml for testing and evaluation purposes. But implementation in C++ could be much more efficient.

The algorithm based on antichains and simulation introduced in [**?**] have been implemented for *tree* automata in library VATA. Description of this library will be placed in next section.

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] [12].

Purposes of VATA library are similar like purposes of this work, so it was decided not to creating brand new library, but makes extension of VATA library 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.

As you can see on picture 4.1, VATA is written in modular way, so it is easy to make extension for finite automata. Thanks to the modularity, any new encoding can share other parts of library like parser or serializer [12].

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

VATA library supports now just *Timbuk* format as input format of *tree* automata [1].

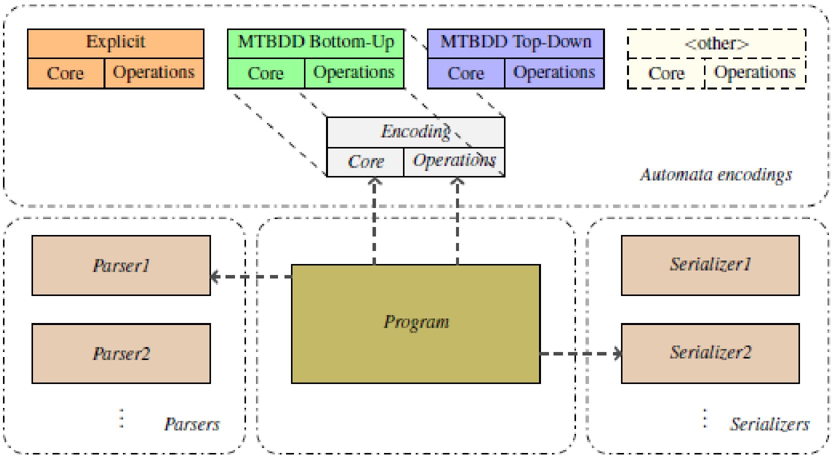

Figure 4.1: The VATA library design. Picture comes from [12]

.

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) [12].

For better performance is used *copy-on-write* technique [12]. 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 [12].

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

Operations

There are supported basic operations like union, intersection, elimination of unreachable states, but also some advance 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 look-up in virtual-method table is very expensive [12]. More details about implementation optimization could be found in [12].

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

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

Table 4.1: Table of some supported operations

4.2.3 Extension for Finite Automata

This work creates extension of VATA library for finite automata in explicit encoding. The main goal is provide operation for language inclusion test of NFA without 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 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. This new extension will use some existing interfaces like simulation computation.

[2] LTS – Labeled Transitions System

Chapter 5

Design

Výsledky

Chapter 6

Implementation

Chapter 7

Experimental evaluation

Chapter 8

Conclusion

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] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *36th FOCS*, Annual Symposium on Foundations of Computer Science, pages 453–462. IEEE Computer Society Washington, 1995.

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

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

[12] 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.

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

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