LIBRARY FOR FINITE AUTOMATA AND TRANSDUCERS

KNIHOVNA PRO KONEČNÉ AUTOMATY A PŘEVODNÍKY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MICHAELA BIELIKOVÁ**
AUTOR PRÁCE

SUPERVISOR **MARTIN HRUŠKA, Ing.**
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

Finite state automata are widely used in the field of computer science such as formal verification, system modelling, and natural language processing. However, the models representing the reality are very complicated and can be defined upon big alphabets, or even infinite alphabets, and thus contain a lot of transitions. In these cases, using classical finite state automata is not very efficient. Symbolic automata are more concise by employing predicates as transition labels. Finite state transducers also have a wide range of application such as linguistics or formal verification. Symbolic transducers replace classic transition labels with two predicates, one for input symbols and one for output symbols. The goal of this thesis is to design a library which can handle different types of symbolic automata to provide a platform for fast prototyping. The main goal of the library is to implement efficient algorithms for symbolic automata processing. The library should additionally include algorithms for symbolic transducers.

Abstrakt

Konečné automaty majú široké uplatnenie v informatike, okrem iných vo formálnej verifikácii, modelovaní systémov a spracovaní prirodzeného jazyka. Avšak modely skutočne reprezentujúce realitu bývajú veľmi komplikované a môžu byť definované nad veľkými, v niektorých prípadoch až nekonečnými, abecedami, a teda môžu obsahovať veľký počet prechodov. V týchto prípadoch nemusí byť je použitie algoritmov na prácu s konečnými automatmi efektívne. Symbolické automaty poskytujú stručnejší zápis tak, že namiesto symbolov v prechodoch používajú predikáty. Konečné prevodníky tiež majú široké uplatnenie, od ligvistiky až po formálnu verifikáciu. Symbolické prevodníky nahradzujú symboly dvojicou predikátov - jeden predikát pre vstupné symboly a jeden pre výstupné. Cieľom tejto práce je návrh knižnice, ktorá vie efektívne pracovať s rôznymi typmi symbolických automatov a umožňuje rýchle prototypovanie nových algoritmov. Hlavným cieľom knižnice je implementovať efektívne algoritmy pre prácu so symbolickými automatmi. Knižnica má taktiež obsahovať algoritmy pre symbolické prevodníky.

Keywords

finite state automata, symbolic automata, transducers, efficient algorithms, formal verification

Klíčová slova

konečné automaty, symbolické automaty, transducery, efektívne algoritmy, formálna verifikácia

Reference

BIELIKOVÁ, Michaela. *Library for Finite Automata and Transducers.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hruška Martin.

Library for Finite Automata and Transducers

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Martin Hruška All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Michaela Bieliková
May 8, 2017

Acknowledgements

Here it is possible to express thanks to the supervisor and to the people which provided professional help (external submitter, consultant, etc.).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Finite automata are used in a wide range of applications in computer science, from regular expressions or formal specification of various languages and protocols to natural language processing [15]. Further applications are hardware design, formal verification or DNA processing. Finite automata are syntactically simple formalism with generality. They found an application in formal verification, for example in checking language inclusion. The field of formal specification use of finite automata to describe a complex process, for example communication between client and server, in a general, formal and concise way. We further introduce finite transducers which are also widely used in computer science. Transducers have application in natural language processing where they can describe phonological rules or form translation dictionaries or in formal verification [6, 13, 15].

While these formalisms are of practical use, when large alphabets are used the number of transitions and therefore the computing demand quickly increases. Furthermore, the common forms of automata and transducers cannot handle infinite alphabets. In practice big alphabets are widely used in natural language processing, where an alphabet must contain all symbols of a natural language. Languages that derive from Latin alphabet such as english usually contain less than 30 symbols, but with the use of diacritic, this number can grow to double. This number further increases in alphabets that have a symbol for every syllable such as Chinese or Japanese. Even larger alphabets may appear in applications in which the symbols are words. Electronic dictionaries often have more than 200K words and even this number is not enough to handle unrestricted texts. In reality, robust syntactic parsers often require an infinite alphabet [13]. In these cases, an extension with predicates replacing elementary symbols can be used. Predicates allow representing a set of symbols with one expression and therefore can reduce the number of transitions. This formalism is known as symbolic automata and transducers. As it will be shown, the most of the operations used on finite automata are easily generalizable for symbolic automata and transducers.

Important operations over automata and transducers, such as language inclusion which is often used in formal verification, have high the upper bound complexity (big O). Other algorithms, such as determinization, can exponentially increase the number of states or transitions. The increase is closely related to the size of the used alphabet. In use cases with big alphabets, this can be partly eliminated by using efficient algorithms for automata processing. This thesis studies two approaches to efficiently checking language inclusion — simulations [9] and antichains [10]. Simulations examine the language preserving relation of each pair of states and then eliminate the states which have the same behaviour for every possible input symbol. This allows reducing the number of states and therefore enables more efficient manipulation with the reduced automaton. Antichains are mostly used in language inclusion and universality checking which can be decided by finding contradictions. The main idea behind antichains is that if we have not found a contradiction in a small set of automaton states, there is not a point of looking for a contradiction in a superset of these states. Including more states in the checked set cannot reduce the accepted language. The superset of an already checked state therefore only widens the accepted language and cannot contain a new contradiction. Formal definitions and algorithms for simulation and antichains as well as their usage in symbolic automata can be found in Chapter 4 of this thesis.

Currently, there are more libraries that deal with some form of symbolic automata. The first ones are AutomataDotNet in C# by Margus Veanes [16], and symbolicautomata in Java by Loris d'Antoni [7]. While these libraries are very efficient and offer many advanced algorithms, such as determinization and minimization and even simulation, they are very complex and have a slow learning curve and therefore are not usable for quick prototyping of new algorithms. Then there is the VATA library[4] in C++, which is a high efficient open source library. VATA offers basic operations, such as union and intersection, and also more complex ones, such as reduction based on simulation [9] or language inclusion checking with antichains optimisation [5]. It can also handle tree automata. It is modular and therefore easily extendible, but since it is written in C++, prototyping in VATA is not easy. FAdo library[1] is a library for finite automata and regular expressions written in Python. Unfortunately, it is a prototype of a library, immature and not well documented. Very fast and efficient library is FSA written in Prolog [2]. FSA offers determinization, minimization, Epsilon removal and other algorithms, and also supports transducers. Version FSA6 of this library extends these the efficiency by allowing predicates to be used in transitions. Unfortunately, Prolog is not so widely used and this library is outdated. Therefore, the goal of this thesis is to design and implement a new library that allows easy and fast prototyping of advanced algorithms for symbolic automata and symbolic transducers. It should allow easy implementation of new operations as well as adding new types of predicates. It should be used for fast implementation and optimisation of advanced algorithms.

Chapter 2 contains theoretical background for the thesis. Various algorithms for symbolic automata are described in Chapter 3. Efficient algorithms for automata are discussed in Chapter 4. Existing libraries for automata are introduced in Chapter 5. The design of created library is described in Chapter 6. Implementation details of symbolic automata library can be found in Chapter 7. Chapter 8 contains data from the experimental evaluation of created library. Summarization and possibilities for future work can be found in Chapter 9.

Chapter 2

Preliminaries

This chapter contains theoretical background for this thesis. First, languages and classic finite automata will be defined, then predicates and operations over them. After the definition of predicates, symbolic automata and transducers are introduced. Proofs are not given but can be found in the referenced literature.[10, 12, 14, 15, 8]

2.1 Languages

Let Σ be an *alphabet* - finite, nonempty set of symbols. Common examples of alphabets include the binary alpabet - $\Sigma = \{0, 1\}$ or an alphabet of lowercase letters - $\Sigma = \{a, b, ..., z\}$.

A *word* or a *string* w over Σ of *length* n is a finite sequence of symbols $w = a_1 \cdots a_n$, where $\forall 1 \leq i \leq n \ : \ a_i \in \Sigma$. An *empty word* is denoted as $\varepsilon \notin \Sigma$ and its length is 0. We define *concatenation* as an associative binary operation on words over Σ represented by the symbol $\cdot$ such that for two words $u = a_1 \cdots a_n$ and $v = b_1 \cdots b_m$ over Σ it holds that $\varepsilon \cdot u = u \cdot \varepsilon = u$ and $u \cdot v = a_1 \cdots a_n b_1 \cdots b_m$. Some strings from binary alphabet $\Sigma = \{0, 1\}$ are for example 00, 111 or 1001011.

Σ^* represents a set of all strings over Σ including the empty word. A *language* $L \subseteq \Sigma^*$ is a set of strings where all strings are chosen from Σ^*. A language over binary alphabet is for example $L = \{0, 01, 10, 11, 111\}$.

In cases where $L = \Sigma^*$, L is called the *universal language* over Σ.

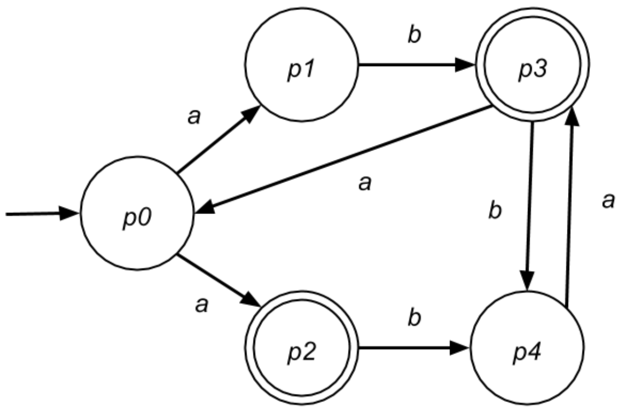
2.2 Finite automata

2.2.1 Nondeterministic finite automaton

A *nondeterministic finite automaton* (NFA) is a tuple $A = (\Sigma, Q, I, F, \delta)$, where:

- Σ is an alphabeth
- Q is a finite set of states
- $I \subseteq Q$ is a nonempty set of initial states
- $F \subseteq Q$ is a set of final states
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. If $q \in \delta(p, a)$ we use $p \xrightarrow{a} q$ to denote the transition from the state p to the state q with the label a.

Figure 2.1: Nondeterministic finite automaton



An example of NFA A is shown in figure 2.1. In this case, $\Sigma = \{a, b\}$, $Q = \{p0, p1, p2, p3, p4\}$, $I = \{p0\}$, $F = \{p2, p3\}$, $\delta = \{(p0, a, \{p1, p2\}),\ (p1, b, \{p3\}),\ (p2, b, \{p4\}),\ (p3, a, \{p0\}),\ (p3, b, \{p4\}),\ (p4, a, \{p3\})\}$. Notice two nondeterministic transitions from $p0$. When the current state of automaton is $p0$ and the input symbol is a, it cannot be deterministically decided whether the next state should be $p1$ or $p2$.

2.2.2 Deterministic finite automaton

A *deterministic finite automaton* (*DFA*) is a special case of an NFA, where $|I| = 1$ and δ is a partial function restricted such that if $\delta(p, a) = q$ DFA cannot contain another transition where $\delta(p, a) = q'$ and $q \neq q'$. A DFA is a tuple $A = (\Sigma, Q, I, F, \delta)$, where:

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

Informally said, a DFA must have exactly one initial state and cannot contain more transitions from one state labelled with the same symbol. An example of DFA A is shown in figure 2.2.

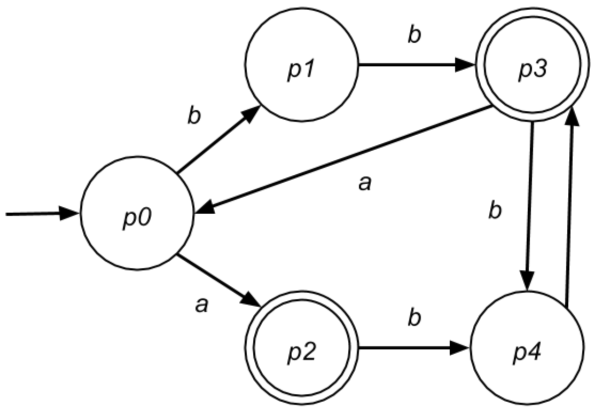
2.2.3 Run of a finite automaton

A *run* of an NFA $A = (\Sigma, Q, I, F, \delta)$ from a state q over a word $w = a_1 \cdots a_n$ is a sequence $r = q_0 \cdots q_n$, where $0 \leq i \leq n$ and $q_i \in Q$ such that $q_0 = q$ and $q_i \xrightarrow{a_{i+1}} q_{i+1} \in \delta$.

The run r is called *accepting* if $q_n \in F$. A word $w \in \Sigma^*$ is called *accepting* if there exists an accepting run from some initial state over w.

An *unreachable* state q of an NFA $A = (\Sigma, Q, I, F, \delta)$ is a state for which there is no run $r = q_0 \cdots q$ of A over a word $w \in \Sigma^*$ such that $q_0 \in I$. It is a state that cannot be reached starting from any initial state.

Figure 2.2: Deterministic finite automaton



An *useless* or *nonterminating* state q of an NFA $A = (\Sigma, Q, I, F, \delta)$ is a state such that there is no accepting run $r = q \cdots q_n$ of A over a word $w \in \Sigma^*$. It is a state from which no final state can be reached.

2.2.4 Language of a finite automaton

Consider an NFA $A = (\Sigma, Q, I, F, \delta)$. The *language* of a state $q \in Q$ is defined as

$$L_A(q) = \{w \in \Sigma^* \mid \textit{ there exists an accepting run of } A \textit{ from } q \textit{ over } w\}$$

Given a pair of states $p, q \in Q$ of an NFA $A = (\Sigma, Q, I, F, \delta)$, these states are language equivalent if:

$$\forall w \in \Sigma^* : A \textit{ run from } p \textit{ over } w \textit{ is accepting} \Leftrightarrow A \textit{ run from } q \textit{ over } w \textit{ is accepting.}$$

The language of a set of states $R \subseteq Q$ is defined as $L_A(R) = \bigcup_{q \in R} L_A(q)$. The language of an NFA A is defined as $L_A = L_A(I)$.

2.2.5 Complete DFA

Complete DFA $A = (\Sigma, Q_C, I_C, F_C, \delta_C)$ is DFA where for any $p \in Q_C$ and every $a \in \Sigma$ exists $q \in Q_C$ such that $p \xrightarrow{a} q \in \delta_C$. Every DFA can be transformed into a complete DFA in two simple steps:

- add a new state to Q, for example $sink \notin Q$ as a nonterminating state
- for every $(q, a) \in Q \times \Sigma$ which is not defined in δ add transition $q \xrightarrow{a} n$ to δ

2.2.6 Well-specified DFA

$Well-specified$ DFA $A = (\Sigma, Q_C, I_C, F_C, \delta_C)$ is DFA where

- Q has no unreachable state
- Q has at most one nonterminating state

2.2.7 Minimal DFA

Minimal DFA $A = (\Sigma, Q, I, F, \delta)$ is a complete DFA where:

- there are no unreachable states
- there is at most one nonterminating state
- there are no two language equivalent states

2.2.8 Finite automata operations

Intersection

Intersection of two NFA $A = (\Sigma, Q_A, I_A, F_A, \delta_A)$ and $B = (\Sigma, Q_B, I_B, F_B, \delta_B)$

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

where δ is defined as

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

The construction yields an automaton with language $L_{A\cap B} = L_A \cap L_B$

Union

Union of two NFA $A = (\Sigma, Q_A, I_A, F_A, \delta_A)$ and $B = (\Sigma, Q_B, I_B, F_B, \delta_B)$ is defined as

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

The construction yields an automaton with language $L_{A\cup B} = L_A \cup L_B$

Determinization

The determinization algorithm transforms any NFA $A = (\Sigma, Q, I, F, \delta)$ into a language equivalent DFA. Algorithm 1 introduced in this thesis is called *powerset construction*. An NFA may have different runs on a word w that use different transitions and therefore may lead to different states. A DFA must have at most one run on a word w. In powerset construction algorithm, we denote Q_w a set of states q such that some run of automaton on word w leads from the initial state to state q. Intuitively, if at least one $q \in Q_w$ is a final state of the NFA, the set Q_w will be the final state of the DFA. The transitions between two sets Q_w and Q'_w will be the union of transitions from each $q \in Q_w$ to each $q' \in Q'_w$.

Complement

Complement of a complete DFA $A = (\Sigma, Q, I, F, \delta)$ is defined as

$$A_C = (\Sigma, Q, I, Q - F, \delta)$$

The construction yields an automaton with language $L_{A_C} = \Sigma^* - L_A$

Algorithm 1: Algorithm for determinization of NFA

```
   Input: NFA $A = (\Sigma, Q, I, F, \delta)$
   Output: DFA $A_D = (\Sigma, Q_D, I_D, F_D, \delta_D)$ where $L_D = L_A$
1  $Q_D, \delta_D, F_D \leftarrow \emptyset$;
2  $\mathcal{W} = \{I\}$;
3  while $\mathcal{W} \neq \emptyset$ do
4      pick $Q'$ from $\mathcal{W}$;
5      add $Q'$ to $Q_D$;
6      if $Q' \cap F \neq \emptyset$ then
7          add $Q'$ to $F_D$;
8      end
9      for $a \in \Sigma$ do
10         $Q'' \leftarrow \bigcup_{q \in Q'} \delta(q, a)$
11         if $Q'' \notin Q_D$ then
12             add $Q''$ to $\mathcal{W}$
13         end
14         add $(Q', a, Q'')$ to $\delta_D$
15     end
16 end
```

Minimization

The minimization algorithm transforms DFA $A = (\Sigma, Q, I, F, \delta)$ into a language equivalent minimal DFA. The idea behind this algorithm is to split the states of DFA into blocks, where a block contains states recognising the same language. The pseudo algorithm for this step called *language partition* can be found in Algorithm 2. After computing language partition, we merge the states of each block into a single state. If all states in the block are final states of the original DFA, this block is final in the resulting minimal DFA. There is a transition (B, a, B') from block B to B' if such a transition exists that $\delta(q, a) = q'$.

2.3 Regular Languages

A language L is *regular* if there existis an NFA $A = (\Sigma, Q, I, F, \delta)$ that $L = L_A$.

2.3.1 Closure Properties

Regular languages are closed under an operation if the operation on some regular languages always results in a regular language.

Closure properties for regular languages include:

- Union: $L = L_1 \cup L_2$.
- Intersection: $L = L_1 \cap L_2$.
- Complement: $L = \overline{L_1}$.
- Difference: $L = L_1 - L_2$.

Algorithm 2: Algorithm for language partition

```
Input: DFA $A = (\Sigma, Q, I, F, \delta)$
Output: language partition $P$
if $F = \emptyset$ or $Q - F = \emptyset$ then
    return $\{Q\}$;
else
    $P \leftarrow \{F, Q - F\}$
end
$\mathcal{W} \leftarrow \{(a, min\{F, Q - f\}\} | a \in \Sigma$ while $\mathcal{W} \neq \emptyset$ do
    pick $(a, B')$ from $\mathcal{W}$;
    for $B \in P$ split by $(a, B')$ do
        replace $B$ by $B_0$ and $B_1$ in $P$;
        for $b \in \Sigma$ do
            if $(b, B) \in \mathcal{W}$ then
                replace $(b, B)$ by $(b, B_0)$ and $(b, B_1)$ in $\mathcal{W}$
            else
                add $(b, min\{B_0, B_1\})$ to $\mathcal{W}$
            end
        end
    end
end
```

- Reversal: $L = \{a_1 \ldots a_n \in \Sigma^* \mid y = a_n \ldots a_1 \in L\}$.
- Concatenation: $L \cdot K = \{x \cdot y \mid x \in L \wedge y \in K\}$.

The first three operations (union, intersection and complement) can be done using finite automata representation for given regular language. Description of these operations over finite automata can be found in 2.2.8. Detailed descriptions, proofs and algorithms for the other operations can be found in the referenced literature [12, 8].

2.3.2 Decidability

A problem about regular language is decidable if such an algorithm exists that can answer the question for every regular language.

Decidable problems for regular languages are:

- *Emptiness* problem: Is language L empty?
- *Membership* problem: Does a particular string w belong to language L?
- *Equivalence* problem: Does language L_1 describe the same language as language L_2?
- *Infiniteness* problem: Is language L infinite?
- *Finiteness* problem: Is language L finite?
- *Inclusion* problem: Is $L_1 \subseteq L_2$?
- *Universality* problem: Is $L = \Sigma^*$?

2.4 Finite transducer

$Finite transducers$ differ from finite automata in transition labels. While finite automata labels contain only one input symbol, in finite transducers both input and output symbols are present. Additionally, transducers have two alphabets — one for input symbols and one for output symbols. Finite transducers can be informally described as translators which when given an input symbol generate an output symbol. This principle has application in a wide range of computer science, such as formal verification or natural language processing. In language processing, transducers can be used for example to create translation dictionaries, where input symbol is a word in one language and the output symbol a word in another language.

A *nondeterministic finite transducer* (NFT) is a tuple $T = (\Sigma, \Omega, Q, I, F, \delta)$, where:

- Σ is an input alphabeth
- Ω is an output alphabet
- Q is a finite set of states
- $I \subseteq Q$ is a nonempty set of initial states
- $F \subseteq Q$ is a set of final states
- $\delta \subseteq Q \times (\Sigma : \Omega) \times Q$ is the transition relation. We use $p \xrightarrow{a:b} q$ to denote the transition from the state p to the state q with the input symbol a, which generates the output symbol b.

An example of a nondeterministic finite transducer is shown in Figure 2.3.

A *deterministic finite transducer* (DFT) must have exactly one initial state and cannot contain more transitions from one state labelled with the same input symbol.

2.4.1 Translation of a finite transducer

A *configuration* of a finite state transducer $M = (\Sigma, \Omega, Q, I, F, \delta)$ is a string vpu where $p \in Q$, u in Σ^* and $v \in \Omega^*$.

Let $vpxu$ and $vyqu$ be two configurations of transducer $M = (\Sigma, \Omega, Q, I, F, \delta)$ where $p, q \in Q$, x, u in Σ^* and $v, y \in \Omega^*$. The M makes a *move* from $vpxu$ to $vyqu$ according to transition r written as $vpxu \vdash vyqu[r]$ or simply $vpxu \vdash vyqu$. We use $\vdash^*$ to denote a sequence of consecutive moves.

A translation $T(M)$ of a finite transducer is:

$$T(M) = \{(x, y) : sx \vdash^* yf, x \in \Sigma^*, y \in \Omega^*, f \in F\}$$

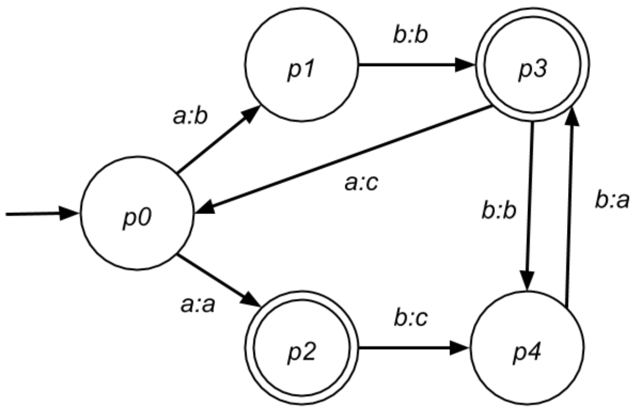
An input language corresponding to $T(M)$ is:

$$L_I(M) = \{x : (x, y) \in T(M) \ for\ some\ y \in \Omega^*\}$$

An output language corresponding to $T(M)$ is:

$$L_O(M) = \{y : (x, y) \in T(M) \ for\ some\ x \in \Sigma^*\}$$

Figure 2.3: Nondeterministic finite transducer



2.5 Predicates

A *predicate* π is a formula representing a subset of Σ. Π denotes a given set of predicates such that, for each element $a \in \Sigma$ there is a predicate representing $\{a\}$, and Π is effectively closed under Boolean operations.

Predicates given in this thesis are inspired by a Prolog library FSA [2]. The semantics of these predicates is:

- $in\{a_1, a_2, \ldots, a_i\}$ represents a subset $\{a_1, a_2, \ldots, a_i\} \in \Sigma$
- $not\_in\{a_1, a_2, \ldots, a_i\}$ represents a subset $\Sigma - \{a_1, a_2, \ldots, a_i\}$

2.5.1 Operations over predicates

Predicates must support conjunction, disjunction and complement. Conjunction and disjunction of *in* and *not_in* predicates can be found in Table 2.1 and Table 2.2 and the complement in Table 2.3.

Table 2.1: Conjunction and of predicates

x	**y**	**x ∧ y**
$in\{a_0, a_1, \ldots, a_n\}$	$in\{b_0, b_1, \ldots, b_m\}$	$in\{\{a_0, a_1, \ldots, a_n\} \cap \{b_0, b_1, \ldots, b_m\}\}$
$in\{a_0, a_1, \ldots, a_n\}$	$not\_in\{b_0, b_1, \ldots, b_m\}$	$in\{\{a_0, a_1, \ldots, a_n\} - \{b_0, b_1, \ldots, b_m\}\}$
$not\_in\{a_0, a_1, \ldots, a_n\}$	$in\{b_0, b_1, \ldots, b_m\}$	$in\{\{b_0, b_1, \ldots, b_m\} - \{a_0, a_1, \ldots, a_n\}\}$
$not\_in\{a_0, a_1, \ldots, a_n\}$	$not\_in\{b_0, b_1, \ldots, b_m\}$	$not\_in\{\{a_0, a_1, \ldots, a_n\} \cup \{b_0, b_1, \ldots, b_m\}\}$

Table 2.2: Disjunction of predicates

x	**y**	**x ∨ y**
$in\{a_0, a_1, \ldots, a_n\}$	$in\{b_0, b_1, \ldots, b_m\}$	$in\{\{a_0, a_1, \ldots, a_n\} \cup \{b_0, b_1, \ldots, b_m\}\}$
$in\{a_0, a_1, \ldots, a_n\}$	$not\_in\{b_0, b_1, \ldots, b_m\}$	$not\_in\{\{b_0, b_1, \ldots, b_m\} - \{a_0, a_1, \ldots, a_n\}\}$
$not\_in\{a_0, a_1, \ldots, a_n\}$	$in\{b_0, b_1, \ldots, b_m\}$	$not\_in\{\{a_0, a_1, \ldots, a_n\} - \{b_0, b_1, \ldots, b_m\}\}$
$not\_in\{a_0, a_1, \ldots, a_n\}$	$not\_in\{b_0, b_1, \ldots, b_m\}$	$not\_in\{\{a_0, a_1, \ldots, a_n\} \cap \{b_0, b_1, \ldots, b_m\}\}$

Table 2.3: Complement of predicates

x	$\neg$ **x**
$in\{a_0, a_1, \ldots, a_n\}$	$not\_in\{a_0, a_1, \ldots, a_n\}$
$not\_in\{a_0, a_1, \ldots, a_n\}$	$in\{a_0, a_1, \ldots, a_n\}$

2.6 Symbolic automata

A *symbolic automaton* A is a tuple $A = (\Sigma, Q, I, F, \Pi, \delta)$, where:

- Σ is an alphabet
- Q is a finite set of states
- $I \subseteq Q$ is a nonempty set of initial states
- $F \subseteq Q$ is a set of final states
- Π is a set of predicates over Σ
- $\delta \subseteq Q \times (\Pi \cup \{\varepsilon\}) \times Q$ is the transition relation.

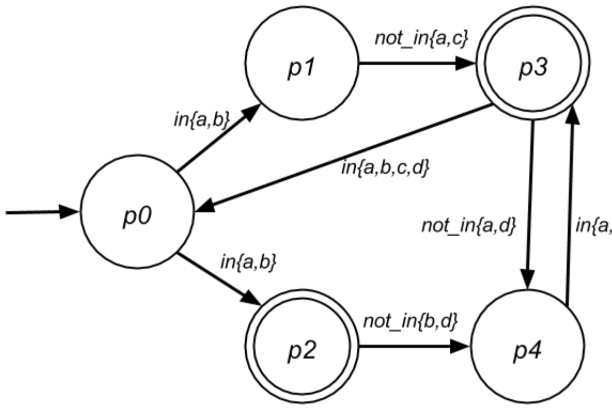
An example of SA A with predicates *in* and *not_in* introduced in 2.5 is shown in Figure 2.4.

Every SA with a finite alphabet can be transformed into a NFA in two steps:

- remove Π
- for each transition $p \xrightarrow{\pi} q \in \delta$:
  - if π is *in* predicate, create a transition $p \xrightarrow{a} q$ for every $a \in \pi$
  - if π is *not_in* predicate, create a transition $p \xrightarrow{a} q$ for every $a \in \Sigma - \pi$

Since every SA can be transformed into NFA, closure and decidability properties of symbolic automata are the same as for finite automata, described in 2.3.1 and 2.3.2. Also, every operation applicable on finite automata can be applied on symbolic automata after transformation to a finite automaton. Fortunately, this transformation can be usually avoided because most of the algorithms for finite automata can be modified to work on symbolic automata. The modified algorithms will be described in 3.

Figure 2.4: Symbolic automaton



2.7 Symbolic transducers

A *symbolic transducer* A is a tuple $A = (\Sigma, \Omega, Q, I, F, \Pi, \delta)$, where:

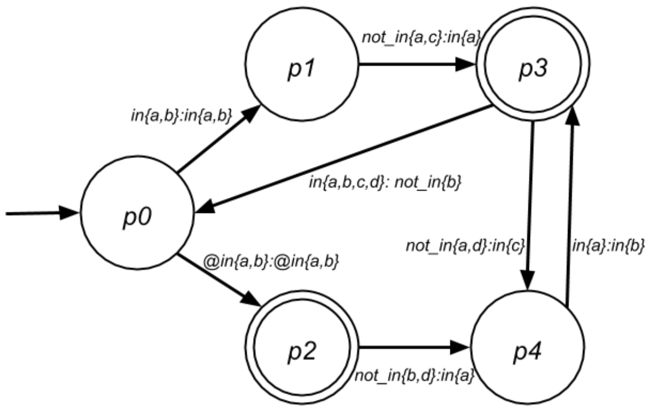
- Σ is an input alphabet
- Ω is an output alphabet
- Q is a finite set of states
- $I \subseteq Q$ is a nonempty set of initial states
- $F \subseteq Q$ is a set of final states
- Π is a set of predicates over Σ
- $\delta \subseteq Q \times (\Pi \cup \{\varepsilon\}) \times (\Pi \cup \{\varepsilon\}) \times Q$ is the transition relation.

A special case of a transition is *identity*. Identity takes an input a and copies it on the output. Identity has a special predicate syntax, for *in* and *not_in* predicates it is denoted by @ in front of the predicate.

An example of a symbolic transducer is shown in Figure 2.5. Transition from $p0$ to $p1$, $p0 \xrightarrow{in\{a,b\}:in\{a,b\}} p1$ would be represented as these 4 transitions in classic transducer: $p0 \xrightarrow{a:a} p1$, $p0 \xrightarrow{a:b} p1$, $p0 \xrightarrow{b:a} p1$, $p0 \xrightarrow{b:b} p1$. The syntax of identity can be seen in the transition from $p0$ to $p2$: $p0 \xrightarrow{@in\{a,b\}:@in\{a,b\}} p1$. In the classic transducer, it would be represented by the transitions $p0 \xrightarrow{a:a} p1$ and $p0 \xrightarrow{b:b} p1$.

Symbolic transducers allow more concise representation on finite transducers. Since symbolic transducers can be transformed to classic finite transducers by creating transitions for each pair of symbols represented by input and output predicates, most operations applicable on finite transducers can be applied on symbolic transducers as well. Fortunately, in many cases, this transformation can be avoided. Algorithms can usually be generalised to work on symbolic transducers, similarly to operations on symbolic automata. The modifications of some of the algorithms is discussed in Chapter 3.

Figure 2.5: Symbolic finite transducer



Chapter 3

Algorithms for symbolic automata and transducers

This chapter describes various algorithms for symbolic automata and transducers. It is shown that most of the algorithms can be easily modified to work with predicates instead of symbols. [6, 15]

Symbolic automata are basically a more general description for finite automata. However, operations applicable on finite automata are also applicable on symbolic automata. In the worst case, these operations would expand symbolic automata to ordinary finite automata then perform the desired operation and transform the resulting automata back to symbolic form. Fortunately, this process is not necessary for most of the algorithms. As will be shown, most of the algorithms can be generalised and used directly on symbolic automata. The same is true for transducers.

3.1 Intersection

The intersection is an powerful operation. In the classic finite automata, the intersection of two NFA $A = (\Sigma, Q_A, I_A, F_A, \delta_A)$ and $B = (\Sigma, Q_B, I_B, F_B, \delta_B)$ is defined as:

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

where δ is defined as:

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

The same approach can be used for symbolic automata, but instead of requiring the symbol $a \in \Sigma$ to be the same in both of the automata, we consider a conjunction of the corresponding predicates in A and B, as shown below. In symbolic transducer we consider a conjunction of both the input predicates and the output predicates.

$$\delta = \{(p_1, q_1) \xrightarrow{\pi_A \wedge \pi_B} (p_2, q_2) | p_1 \xrightarrow{\pi_A} p_2 \in \delta_A \wedge q_1 \xrightarrow{\pi_B} q_2 \in \delta_B\}$$

3.2 Union

Union of symbolic automata can be computed the same way as for classical finite automata which was described in 2.2.8.

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

3.3 Determinization

In classic determinization algorithm, we use subsets of states. Each subset is a set in the resulting deterministic machine. To compute transitions leaving a given subset D, we compute for each symbol σ a set of states Q such that $d \in D$, $q \in Q$ and $(d, \sigma, q) \in \delta$.[14]

In the case of predicates, transitions leaving the given subset might overlap. This situation cannot happen in deterministic automata and thus we must create nonoverlapping transitions. For transitions labeled with predicates P_1 and P_2, we must create the transitions labelled with following predicates: $P_1 \wedge \overline{P_2}$,$P_1 \wedge P_2$, $\overline{P_1} \wedge P_2$. This process is called *getting exclusive predicates* for a group of states and can be found in Algorithm 3.

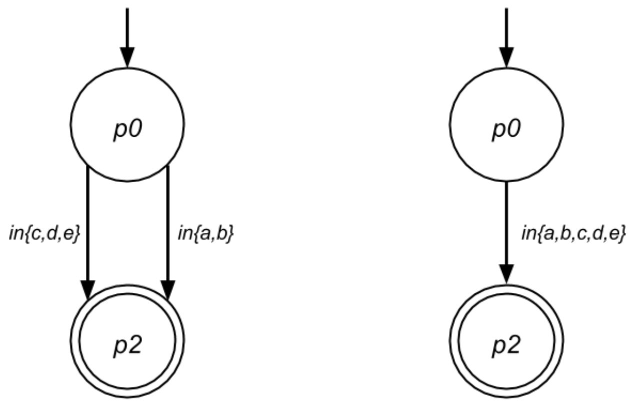
Algorithm 3: Getting exclusive predicates for a group of states

```
   Input: NSA $A = (\Sigma, Q, I, F, \Pi, \delta)$, group of states $Q' \in Q$
   Output: exclusive predicates for $Q'$
 1 $\Pi_e \leftarrow \emptyset$;
 2 for each $q \in Q'$ do
 3     $\Pi' = \{\pi | (q, \pi) \in \delta\}$;
 4     $C \leftarrow$ all boolean combinations of $\pi \in \Pi$;
 5     $\Pi_e = \Pi_e \cup C$
 6 end
 7 return $\Pi_e$
```

Algorithm 4: Algorithm for determinization of SA

```
   Input: NSA $A = (\Sigma, Q, I, F, \Pi, \delta)$
   Output: DSA $A_D = (\Sigma, Q_D, I_D, F_D, \Pi, \delta_D)$ where $L_D = L_A$
 1 $Q_D, \delta_D, F_D \leftarrow \emptyset$;
 2 $\mathcal{W} = \{I\}$;
 3 while $\mathcal{W} \neq \emptyset$ do
 4     pick $Q'$ from $\mathcal{W}$;
 5     add $Q'$ to $Q_D$;
 6     if $Q' \cap F \neq \emptyset$ then
 7         add $Q'$ to $F_D$;
 8     end
 9     $\Pi_e \leftarrow$ get exclusive predicates for $Q'$;
10     for $\pi \in \Pi_e$ do
11         $Q'' \leftarrow \bigcup_{q \in Q'} \delta(q, \pi)$
12         if $Q'' \notin Q_D$ then
13             add $Q''$ to $\mathcal{W}$
14         end
15         add $(Q', \pi, Q'')$ to $\delta_D$
16     end
17 end
```

Figure 3.1: Minimization cleanup



3.4 Minimization

In Hopcroft's minimization algorithm [11], we repeatedly refine subsets of states by considering a pair of state and symbol, revealing that an existing subset must be split. The algorithm ends when such a pair no longer exists. The resulting automaton is minimal in the number of states. The algorithm for minimization of classic finite automata was given in 2.2.8.

However, the resulting automaton might not be minimal in the number of transitions. This is caused by the fact that the same transition can be expressed in multiple ways. For example, a transition $p0 \xrightarrow{in\{a,b,c\}} p1$ could also be represented by two transitions: $p0 \xrightarrow{in\{a,b\}} p1$ and $p0 \xrightarrow{in\{c\}} p1$. Therefore, as the final step of minimization, we must perform a cleanup that joins all transitions from state p to state q into one. After this, the resulting automaton will be minimal in the number of transitions. An example of cleanup is shown in Figure 3.1. On the left are the transitions before cleanup and on the right after cleanup.

3.5 Optimization

Previous algorithms which require the predicates to not overlap produce a lot of combinations of predicates. This way the resulting automata might significantly grow in the number of transitions. This can be avoided by using simple optimisations.

3.5.1 Satisfiability

A predicate P is *satisfiable*, when it is true for at least one symbol a where $a \in \Sigma$. Unsatisfiable transitions can be removed from the automaton because they will never be used.

An example of unsatisfiable predicate is $in\{a, b\} \wedge in\{c, d\}$. A transition labeled with this predicate will never be used because a symbol, that is in set $\{a, b\}$ and also in set $\{c, d\}$ does not exist. Thus, this transition can be removed from the automaton without affecting the language accepted by the automaton.

3.5.2 Removing useless states

This optimisation is not dependent on the use of predicates and can be used in classic finite automata as well. *Useless states* the states, from which no final state can be reached. All useless states can be removed from the automaton without affecting the accepted language. All transitions $p \xrightarrow{a} q$ where p is an useless state can also be removed. If the original automaton contains at least one useless state, the resulting automaton will be smaller in the number of states and possibly in the number of transitions.

Chapter 4

Efficient algorithms for automata and transducers

In this chapter efficient algorithms for automata processing, that allow reduction of automata size in the number of states, such as simulations and antichains, are discussed. This thesis covers only the basics, more complex proofs and theories can be found in the referenced literature [9, 10, 13].

In many operations with finite automata, we need a minimised version of the automaton. However, the simplest minimising algorithm demands the automaton to be determinized first. Determinization is very inefficient because in the worst cases the size of the automaton can exponentially increase. To avoid this increase, NFA can be reduced using equivalence or simulation relations. These relations allow merging states that recognise the same language and therefore reducing the size of NFA without determinization. A reduction based on simulations usually yields an automaton that is smaller than minimal DFA but not deterministic.

4.1 Reduction and equivalence

Let $A = (\Sigma, Q, I, F, \Pi, \delta)$ be an NFA. Equivalence relation $\equiv_R \subseteq Q \times Q$ is defined as:

- $\equiv_R \cap (F \times (Q - F)) = \emptyset$
- for any $p, q \in Q, a \in \Sigma, (p \equiv_R q \Rightarrow \forall q' \in \delta(q, a), \exists p' \in \delta(p, a), p' \equiv_R q')$

The first condition simply means that a final state cannot be equivalent to a nonfinal state. The second condition means that two states p and q are equivalent when for every symbol a such that transition $p \xrightarrow{a} p' \in \delta$ a transition $q \xrightarrow{a} q' \in \delta$ must exist and the states p' and q' must also be equivalent.

Symmetrically, a relation $\equiv_L$ can be defined over an reversed automaton. By reversed automaton is meant an automaton, in which transition have been reversed by the rule $q \in \delta_r(p, a)$ if $p \in \delta(q, a)$.

An automaton can be reduced using both equivalencies, but the automaton reduced by $\equiv_R$ and the automaton reduced by $\equiv_L$ do not have to be equivalent.

Implementation of reduction is another interesting problem. Implementing reduction directly by the definition would lead to exponential rise of used memory space because computing equivalence for one state leads to computing equivalence for all the following ones. The computation would recursively check all the following states until it reaches a state with no transitions or finds a contradiction. This strategy is not usable for big automata because recursive checking of a big number of states exponentially raises used memory space and also run time of the algorithm. The more efficient method could be checking the equivalence in the reversed order, by checking the predecessors of a state.

After computing equivalence, the algorithm to reduce the automaton A is trivial: it simply merges all state in the same equivalence class into one and modifies the transitions accordingly.

4.2 Reduction and simulations

A better reduction can be obtained by using simulations instead of equivalence. The definition of simulation $\preceq_R \subseteq Q \times Q$ is similar to equivalence:

- $\preceq_R \cap (F \times (Q - F)) = \emptyset$
- for any $p, q \in Q, a \in \Sigma, (p \preceq_R q \Rightarrow \forall q' \in \delta(q,a), \exists p' \in \delta(p,a), p' \preceq_R q')$

As in the case with equivalencies, $\preceq_L$ can be created using the reversed automaton. If $p \preceq_R q$, then $L_R(p) \subseteq L_R(q)$ and if $p \preceq_L q$ then $L_L(p) \subseteq L_L(q)$.

The reduction using simulations is more complicated than the reduction using equivalencies. We can merge two states p and q as soon as any of the conditions is met:

1. $p \preceq_R q$ and $q \preceq_R p$
2. $p \preceq_L q$ and $q \preceq_L p$
3. $p \preceq_R q$ and $p \preceq_L q$

However, after merging the states according to conditions 1 or 2, relations $\preceq_R$ and $\preceq_L$ must be updated so that their relation with the languages L_R and L_L is preserved. For instance, if merged state of p and q is denoted q, we must remove from $\preceq_R$ any pairs (q, s) for which $p \not\preceq_R s$. Merging according to condition 3 does not require any update.

Pseudocode for efficient computation of simulations is given in Algorithm 5. The algorithm works correctly only on complete NFAs. In this algorithm, $card(n)$ denotes the number of items in set n. The result of this algorithm is $\not\preceq_R$, which is a complement of $\preceq_R$. $\not\preceq_R$ is defined as:

- $(F \times (Q - F)) \subseteq \not\preceq_R$
- for any $p, q \in Q, a \in \Sigma, (\exists p' \in \delta(p,a), \forall q' \in \delta(q,a), p' \not\preceq_R q' \Rightarrow p \not\preceq_R q)$

Since $\not\preceq_R$ is a complement of $\preceq_R$, this algorithm returns all pair of states (p, q) in which q does not simulate p. The same algorithm can be used for computing $\not\preceq_L$ when it is applied on the reversed automaton.

Further in this thesis, $\preceq$ is used as a shorthand for $\preceq_R$.

Algorithm 5: Algorithm for automata reduction with simulations

```
   Input: complete NFA $A = (\Sigma, Q, I, F, \delta)$
   Output: $\not\preceq_R$
 1 for each $q \in Q$ and $a \in \Sigma$ do
 2     compute $\delta^r(q, a)$ as an linked list;
 3     compute $card(\delta(q, a))$;
 4 end
 5 initialize all $N(a)$s with 0s;
 6 $\omega := \emptyset$;
 7 $\mathcal{C} := NEW\_QUEUE()$;
 8 for each $f \in F$ do
 9     for each $q \in Q - F$ do
10         $\omega := \omega \cup \{(f, q)\}$;
11         $ENQUEUE(\mathcal{C}, (f, j))$;
12     end
13 end
14 while $\mathcal{C} \neq \emptyset$ do
15     $(i, j) := DEQUEUE(\mathcal{C})$ ;
16     for each $a \in \Sigma$ do
17         for each $k \in \delta^r(j, a)$ do
18             $N(a)_{ik} := N(a)_{ik} + 1$;
19             if $N(a)_{ik} == card(\delta(k, a))$ then
20                 for $l \in \delta^r(i, a)$ do
21                     if $(l, k) \notin \omega$ then
22                         $\omega := \omega \cup \{(l, k)\}$;
23                         $ENQUEUE(\mathcal{C}, (l, k))$
24                     end
25                 end
26             end
27         end
28     end
29 end
```

4.3 Antichains

The antichains algorithm [5] described in pseudocode in Algorithm 6 can be used for language inclusion checking of finite automata. The textbook approach to language inclusion checking requires both automata to be determinized first. This approach is very inefficient when working with big alphabets and automata. The antichain algorithm allows us to skip the complete determinization. While checking if $L_A \subseteq L_B$, the automaton A is not determinized at all and the automaton B is determinized gradually. If $L_A \nsubseteq L_B$, the algorithm stops when finding the first contradiction, so the automaton B is not completely determinized. The antichains algorithm is also efficient if $L_A \subseteq L_B$ because states which are not necessary to explore are pruned out based on the simulations relation.

We define an antichain 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. $Macrostate$ is defined as a subset of states from Q. For two macrostates P and R of a NFA is $R \preceq^{\forall\exists} P$ shorthand for $\forall r \in R \,.\, \exists p \in P : r \preceq p$. A product state (p, P) of a NFA $A \cap B_{det}$ is witness, if p is final in automaton A and P is not final in automaton B_{det}.

The antichains algorithm tries to find a final state of the product automaton $A \cap \overline{B_{det}}$ while not exploring the states when not necessary. The algorithm explores pairs (p, P) where $p \in Q_A$ and $P \subseteq Q_{B_{det}}$. The automaton B is gradually determinized while constructing $Post(p, P) := \{(p', P') \mid \exists a \in \Sigma : p \xrightarrow{a} p' \in \delta_A, P' = \{p'' \in Q_B \mid \exists p''' \in P : p''' \xrightarrow{a} p'' \in \delta_B\}\}$.

The algorithm derives new states from the product automaton transitions and inserts them to the set $Next$ for further processing. Once a product state from $Next$ is processed it is moved to the set of checked pairs *Processed*. $Next$ and *Processed* keep only minimal elements with respect to the ordering given by $(r, R) \sqsubseteq (p, P)$ iff $r = p \wedge R \subseteq P$. If there is a pair (p, P) generated and there is $(r, R) \in Next \cup Processed$ such that $(r, R) \sqsubseteq (p, P)$, we can skip (p, P) and not insert it to $Next$ for further search.

An improvement of the antichains algorithm is based on simulations. We can stop the search for a pair (p, P) if one of the following conditions is met:

- there exists some already visited pair $(r, R) \in Next \cup Processed$ such that $p \preceq r \wedge R \preceq^{\forall\exists} P$
- there is $p' \in P$ such that $p \preceq p'$

This optimisation is at the lines 12–15 in the pseudo code. The basic explanation is that if the algorithm encounters a macrostate R which is a superset of already checked macrostate P, there is no need to continue the search for this one. If no contradiction was found in the macrostate P, then it cannot be found in R since R is bigger and widens the number of accepted words.

Another optimisation is based on a different principle. It comes from observation that $L(A)(P) = L(A)(P \backslash \{p_1\})$ if there is some $p_2 \in P$ such that $p_1 \preceq p_2$ and $p_1 \neq p_2$. Since P and $P \backslash \{p_1\}$ have the same language, if a word is not accepted from P, it is not accepted from $P \backslash \{p_1\}$ either. Also, if all words from Σ^* are accepted from P, they are also accepted from $P \backslash \{p_1\}$. Therefore, it is safe to replace the macrostate P with macrostate $P \backslash \{p_1\}$. This optimisation is applied by the function *Minimize* at the lines 5 and 8 in the pseudocode.

Algorithm 6: Language inclusion checking with antichains and simulations

```
Input: NFAs $A = (\Sigma, Q_A, I_A, F_A, \delta_A)$ , $B = (\Sigma, Q_B, I_B, F_B, \delta_B)$
A relation $\preceq$ over $A \cup B$ that imply language inclusion.
Output: TRUE if $L_A \subseteq L_B$. Otherwise FALSE.
1  if there is a witness product-state in $\{(i, I_B) \mid i \in I_A\}$ then
2      return FALSE;
3  end
4  $Processed$:=$\emptyset$;
5  $Next$:= $\{(s, Minimize(I_B)) \mid s \in I_A\}$;
6  while $Next \neq \emptyset$ do
7      Pick and remove a product-state $(r, R)$ from $Next$ and move it to $Processed$;
8      foreach $(p, P) \in \{(r', Minimize(R')) \mid (r', R') \in Post(r, R)\}$ do
9          if $(p, P)$ is a witness product-state then
10             return FALSE;
11         else
12             if $\nexists p' \in P$ s.t. $p \preceq p'$ then
13                 if $\nexists (x, X) \in Processed \cup Next$ s.t. $p \preceq x \wedge X \preceq^{\forall\exists} P$ then
14                     Remove all $(x, X)$ from $Processed \cup Next$ s.t. $x \preceq p \wedge P \preceq^{\forall\exists} X$;
15                     Add $(p, P)$ to $Next$;
16                 end
17             end
18         end
19     end
20 end
21 return TRUE;
```

4.4 Antichains and simulations for symbolic automata

The implementation of antichains algorithm for symbolic automata is similar than for finite automata, described in Algorithm 6. This algorithm does not directly work with automata alphabet of transition labels, and as the difference between classic finite automata is only in the transition labels, the pseudocode may be implemented in the same way. Although the main part of the algorithm stays the same, some alteration must be made to the functions used in antichain algorithms.

The first notable difference is in the implementation of simulations relation. The algorithm derives from previously introduced Algorithm 5. It is based on counting the number of transitions from states through alphabet symbols. Since the labels in symbolic automata represent sets of symbols, these computations are implemented by looping through all alphabet symbols and checking if they belong to a predicate. The pseudocode for this alternation can be seen in the fragment of code in Algorithm 7.

Algorithm 7: Transformation of simulations algorithm for symbolic automata

```
1 for $a \in \Sigma$ do
2     for $label$ in $transitions[state]$ do
3         if $label.has\_letter(a)$ then
4             process the transition
5         end
6     end
7 end
```

The same principle is reused in computation of *Post* over symbolic automata as can be seen in Algorithm 8. Each symbol from the alphabet is checked in both automata transitions and the resulting pairs of states and superstates form the *Post* relation.

Algorithm 8: Computing *Post* over symbolic automata

```
1 for $a \in \Sigma$ do
2     for $label$ in $transitions[state]$ do
3         if $label.has\_letter(a)$ then
4             add $transitions[state][label]$ to $Post$
5         end
6     end
7 end
```

The used modifications of antichains algorithm for symbolic automata yield correct results, but they are not very efficient for automata with a big alphabet and a small number of transitions since each alphabet symbol must be checked. Further optimisation of these functions specifically for symbolic automata can make the created library faster and more efficient. These modifications might also be problematic if the library would be altered to support infinite alphabets in the future since looping through all alphabet symbols would not be possible.

Chapter 5

Existing Libraries for Automata

This chapter contains a summary of some of the existing libraries, that deal with finite or symbolic automata. For every chosen library, implementation language, types of supported automata and other details are given. The real number of automata libraries is much bigger, and the inspection of all of them is over the scope of this thesis. Therefore, only the libraries that are related to this thesis, with their design or some used concept, are mentioned.

5.1 AutomataDotNet

AutomataDotNet [16] is a library by Margus Veanes written in .NET framework in C#. It supports algorithms for regular expressions, automata, and transducers. AutomataDotNet can work with symbolic automata where characters are replaced with character predicates, which can be supported by an SMT solver as a plugin. Some of the high efficient algorithms for automata processing are implemented, such as simulations used for automata reduction.

AutomataDotNet is a very complex library that can handle many types of automata from classic finite automata and transducers to tree automata. A big advantage is a possibility to use an SMT solver for predicates. Unfortunately, this library is complex and for extending it many modules and their connections and interactions have to be studied. Because of this, the library is not easly modifiable and thus cannot be used for fast prototyping.

5.2 symbolicautomata

Symbolicautomata [7] is a library by Loris d'Antoni written in Java. The library can handle symbolic automata and algorithms over them such as intersection, union, equivalence and minimization. The predicates supported by symbolicautomata by default are character intervals. For example [a–z] represents every symbol in the interval from a to z. Symbolicautomata is complex. It includes support for symbolic pushdown automata and symbolic transducers. Furthermore, some of the high efficient algorithms for automata processing such as simulations are implemented.

Symbolicautomata provides advanced algorithms for automata processing. Due to complexity, the learning curve of this library is high. The next disadvantage of this library is Java which may be efficient for advanced algorithms but which is not usable for easy and fast prototyping of new algorithms.

5.3 VATA

Libvata [4, 17] is a highly optimised non-deterministic finite tree automata library implemented in C++. It uses the most advanced techniques for automata processing with the focus on using the available computation power as efficiently as possible. VATA can deal with classic finite automata and transducers as well as both explicit and semi-symbolic encoding of tree automata.

VATA has a modular design so the various encodings for automata can be easily added as long as they respect a defined interface. Different encodings may implement different sets of operations. Generally, VATA offers basic operations, such as union and intersection, and also more complex ones, such as reduction based on simulation or language inclusion checking with antichains optimisation. For some operations, such as language inclusion, multiple implementations are available.

VATA is a complex and well optimised library with main focus on tree automata, which can be easily extended because of its modular design, but C++ is not really a good language for fast prototyping. Moreover, VATA does not currently support symbolic automata.

5.4 FAdo

FAdo [1] is a Python library focusing on finite automata and other models of computation. Because this library can work with both regular expressions and automata, it offers only basic algorithms. It can perform conversions between nondeterministic and deterministic automata, as well as conversion between automata and regular expression. FAdo can also transfer automata to simple graphic representations.

Regular expressions, DFA and NFA are implemented as Python classes. Transducers are also available. Elementary regular languages operations as union, intersection, concatenation, complementation and reverse are implemented for each class. For operations like determinization and minimalization, it is possible to choose from more methods, such as Moore, Hopcroft, and some incremental algorithms. Some of these algorithms are implemented in C for higher efficiency.

The biggest advantage of FAdo library is implementation in Python. Python is language that is easy to use and therefore has the potential to be used for fast prototyping of new efficient algorithms. Unfortunately, FAdo library is still immature, more in the state of a prototype, and not very well documented. Extending this library to support symbolic automata would be hard because of the lack of good documentation of its classes, interfaces and inner communication.

5.5 FSA

FSA [2] is a fast library written in Prolog that can handle classic finite automata and transducers as well as symbolic. Predicates *in* and *not_in* which are used in this thesis and will be implemented in the created library are inspired by FSA.

FSA offers determinization, minimization, Epsilon removal and other algorithms, and also supports transducers. Version FSA6 of this library extends these by allowing predicates to be used in transitions. Unfortunately, this library is outdated. Also, Prolog is not widely used and therefore using this library for fast and easy prototyping is not possible.

Chapter 6

Design

This chapter contains a description of library design. The library modules and their connections are discussed.

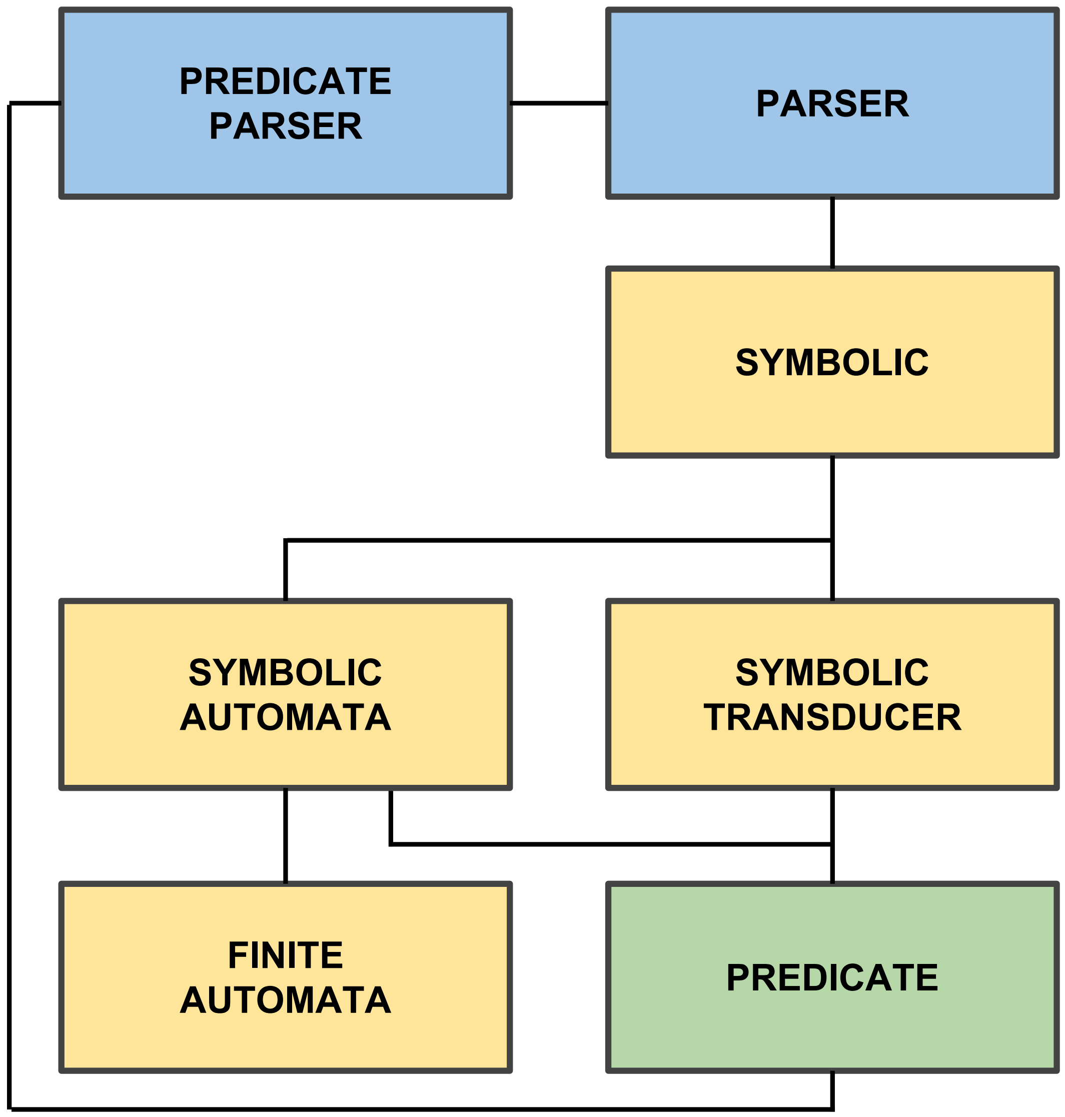
The library is designed to allow easy and intuitive manipulation with symbolic automata and transducers. Basic operations should be available for each supported type of automata. Additionally, some state-of-the-art algorithms for finite automata should be implemented. Adding new operations should be easy and the library will be used for prototyping of new algorithms. None of the operations should be dependent on used predicate type. Adding new types of predicates should be allowed as long as they respect a defined interface.

6.1 Design of the library

The library consists of multiple modules, as is shown in Figure 6.1. The lines in the Figure signifies that the connected modules interact with each other.

- `Parser` is used for automata and transducer parsing.
- `Predicate parser` is called when `Parser` finds a predicate. `Predicate parser` should parse the predicate and transform it into a `Predicate` object. The object is then returned and `Parser` stores it in the automaton object.
- `Symbolic` is a class for operations that are the same for symbolic automata and symbolic transducers. This includes the operations that process the automata independently on the transition labels such as intersection or union.
- `Symbolic automata` is a class for operations over symbolic automata. It includes basic operations such as determinization and minimization as well as efficient and advanced ones, such as simulations and antichain.
- `Symbolic Transducer` contains implementation of operations over symbolic transducers such as composition or application on NFA.
- `Finite automata` is a class in which some algorithms are optimised for classical finite automata.

Figure 6.1: Library Design



6.2 Predicates

In the created library, predicates represent the transition labels. The library supports two different types of predicates by default. When adding new types of predicates, predicate class and parser must be provided. As most of the implemented operations work directly with transition labels, the new predicates must implement the defined interface for the library to work correctly.

6.2.1 Default predicates

Predicates provided by the library as default are:

- *in* and *not_in* predicates, introduced in 2.5
- *letter* predicates, representing symbols used in classic finite automata operations

6.2.2 Predicate interface

There are 5 operations that are needed for the implemented algorithms:

- conjunction — conjunction of two predicates
- disjunction — disjunction of two predicates
- negation — negation of a predicate
- satisfiability — check whether the predicate is satisfiable
- has_letter — check if a symbol belongs to the set represented by the predicate

Conjunction and *disjunction* are needed for basic algorithms such as intersection and union. Algorithms such as determinization and minimisation also need *complement* to be implemented. $Satisfiability$ is used in optimisations described in 3.5. The operation *has_letter* is used in advanced algorithms such as computing simulations relation and the antichains algorithm.

Chapter 7

Implementation

This chapter provides the description of the created library *symboliclib*. The basic information about the library is given first. Then, information about the inner representation of different types of automata is given. Finally, implementation of various algorithms is discussed.

7.1 Symboliclib

Symboliclib is a library implemented in Python 3. It supports finite and symbolic automata as well as finite and symbolic transducers. *Symboliclib* covers basic algorithms for automata processing such as union, intersection, determinization or minimization. Some of the more advanced efficient algorithms such as language checking using simulations or the antichains algorithm are also implemented.

The main goal of the library is to be easily modifiable and extendable and allow fast prototyping of new algorithms. This idea is reflected by the modular design of the library described in Chapter 6. Adding new types of predicates is simple as long as they implement the predefined interface. Usage of SMT solvers is also possible by creating a middle-layer that converts the interface of SMT solver to the interface required by *symboliclib*.

The default input format of *symboliclib* is Timbuk [3] which is so far the only supported format. The library supports conversion of symbolic automata to classic finite automata and serialisation of automata back to Timbuk text format.

The predicates supported by default are *in* and *not_in* predicated inspired by FSA library[2] and *letter* predicates which represent symbols used in classic finite automata and transducers. Some of the algorithms which can be optimized to work on finite automata more efficiently than on symbolic automata are implemented for both types of automata separately.

7.1.1 Command line interface

Symboliclib provides a simple command line interface written in bash. CLI takes a name of the desired operations and path to automata files as arguments. It should cover the most used automata operations as well as different algorithms for inclusion checking described in 7.4.

7.2 Transducers support

Symboliclib library has a basic support for transducers including operations such as union, intersection, composition or running transducer on NFA. Both classic finite and symbolic transducers are supported, but unlike automata, operations are not implemented separately for classic and symbolic transducers. Transducers containing epsilon transitions are not supported, but used data structures and already implemented operations allow to add this support in the future.

Informally said, the only difference between automata and transducers input format and processing are the transition labels. Where one predicate is enough for automata to model configurations of systems, transducers use two predicates to model system behaviour. This difference is handled by *symboliclib* module `Transducer predicate`.

`Transducer predicate` provides a layer for label processing by executing predicate operations on both input and output predicates synchronously. All operations mentioned in 6.2.2 are implemented by `Transducer predicate` so that they process both input and output predicates. This way, the interface for automata and transducer labels processing is the same and therefore algorithms such as intersection and union can be generalised to work on both automata and transducers.

7.3 Inner representation of automata

Inner representation of symbolic and finite automata and transducers is a Python class with the same basic attributes. The library does not support different input and output alphabets for transducers. The classes additionally include some advanced attributes used for optimisation such as *determinized* or *reversed* which represent the determinized and reversed version of the automaton. When determinizing or reversing, the library first checks whether these attributes are non-empty. If they are present, the desired modification of the automata is returned without the need to perform the operation again.

Attributes of the automata such as alphabets and initial, final and all states are represented by a set. This way it is ensured that the same state or symbol is not saved redundantly two times. As an optimisation, after parsing an automaton *symboliclib* removes all nonterminating and useless states and their respective transitions as described in 7.5.

For easy manipulation with automata, transitions are saved in a list of dictionaries. The key of the dictionary is a left-handed side state of a transition and the value is another dictionary. In this dictionary, the key is a predicate and the value is a list of right-handed states.

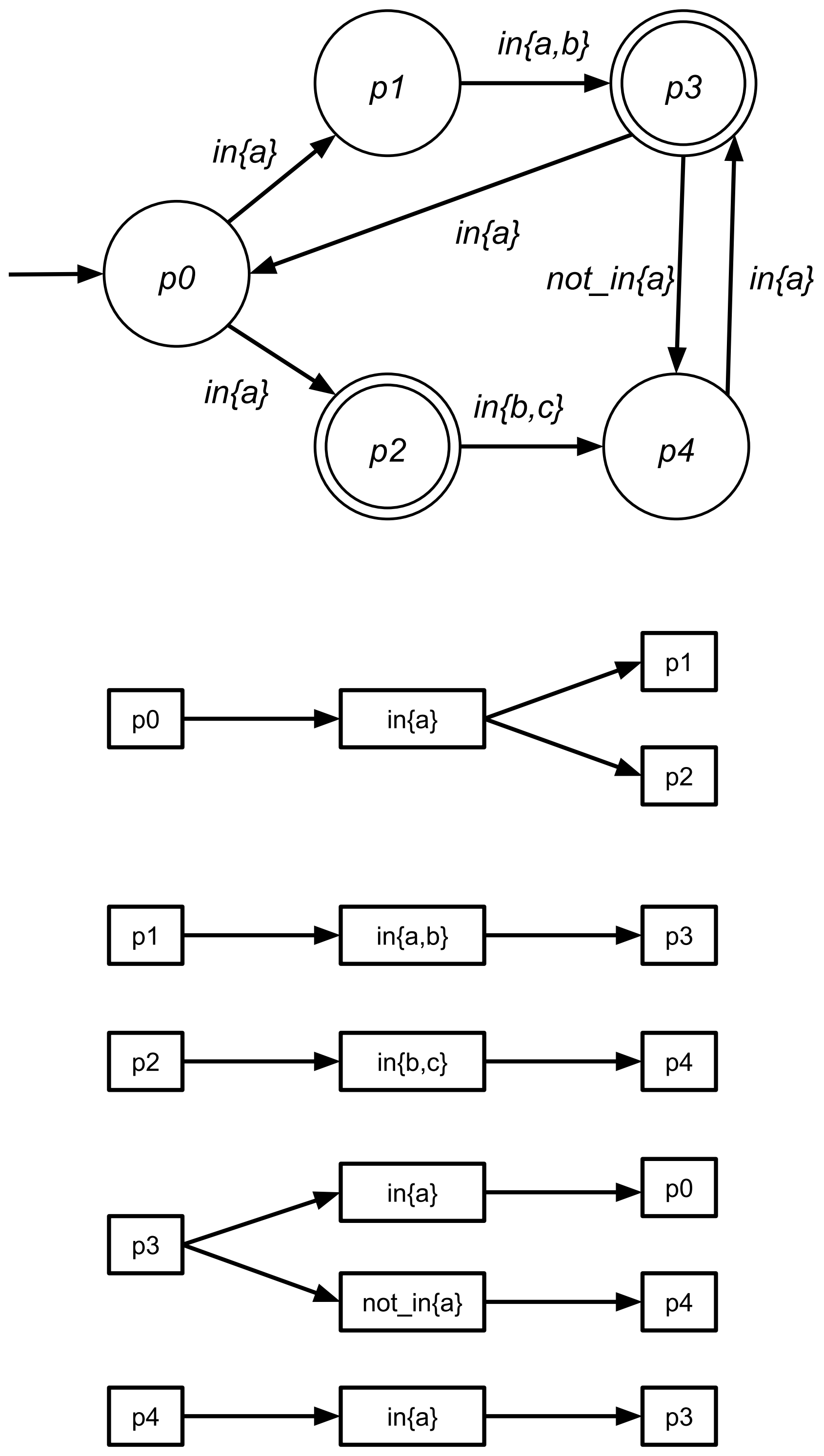
For transitions $p0 \xrightarrow{in\{a,b\}} p1$, $p0 \xrightarrow{in\{a,b\}} p2$ and $p1 \xrightarrow{in\{b\}} p2$ the structure would be:

[{'p0': $\{in\{a,b\}$: [p1,p2]$\}\}, \{$'p1': $\{in\{b\}$: [p2]$\}$}]

A schematic example of automata transitions data structure can be found in Figure 7.1.

For an easier automata manipulation for beginners in programming or automata theory, a function *get_math_format* was added. This function returns automaton in a dictionary containing items *alphabet*, *states*, *initial*, *final* and *transtions*. This way the automata can be manipulated in a more intuitive and direct way without the need to preserve consistency of additional automata attributes.

Figure 7.1: Automata transitions data structure



7.4 Algorithms

Most of the algorithms in *smyboliclib* (intersection, union, determinization, minimisation, epsilon rules removing, etc.) are based on [8].

Some of the operations such as intersection or simulations are separately implemented and optimised for classic finite automata. This allows avoiding operations on sets like intersection and union where a simple equality checking is sufficient. Generally, these operations should run faster on classic finite automata than on symbolic automata.

Symboliclib implements three different algorithms for language inclusion checking which is a widely used operation in formal verification:

7.4.1 Inclusion checking by mathematical definition

Algorithm *is_included_simple* is based on a notion that:

$L(A) \subseteq L(B)$ iff $L(A) \cap \neg L(B) = \emptyset$

The algorithm first creates a complement of B, then constructs an intersection of this complement with A and afterwards checks if the intersection is empty. Since the implementation of complement in *symboliclib* requires the automata to be determinized first, this algorithm is not efficient for big automata.

7.4.2 Classic inclusion checking

Algorithm *is_included* is based on [8], where both automata are determinized first. Then we look for such a pair of states that q_A is final in A, q_B is not final in B and both q_A and q_B can be reached by the same input symbols sequence. If such a pair is found, $L(A) \not\subseteq L(B)$, else $L(A) \subseteq L(B)$.

This algorithms is further used in *symboliclib* implementation of automata equality checking where both $L(A) \subseteq L(B)$ and $L(B) \subseteq L(A)$ must be *true*.

7.4.3 Inclusion checking using simulations and antichains

The basic idea behind the algorithm *is_included_antichain* is the same as in 7.4.2, with antichain optimisations described in 4.3. First, the simulations are computed and then we search for a contradiction in the same way as in 7.4.2, eliminating some of the pairs before checking them based on computed simulations relation.

7.5 Optimizations

Symboliclib includes some optimisations for better performance and faster execution.

After parsing an input automaton, all states are checked. If they are nonterminating or unavailable, they are removed from the automaton and so are all their corresponding transitions. By removing nonterminating and unavailable states, the language accepted by the automaton is not affected, but the size of the automaton may be decreased which leads to faster execution of operations.

Next optimisation of automata size is merging transition labels into one when possible. If an automaton contains transitions $p0 \xrightarrow{in\{a\}} p1$, $p0 \xrightarrow{in\{b\}} p1$, these are merged and saved as $p0 \xrightarrow{in\{a,b\}} p1$. This operation reduces the number of transitions of the automaton. This reduction is especially efficient when the automaton was transformed from classic FA to symbolic, in which case each predicate represents only one symbol.

Operations over automata such as inclusion or determinization can be usually implemented two ways. The first one computes a product of all automata states and studies connections between resulting state pairs. This may generate states and transitions that are either unavailable or nonterminating and therefore wastes computing power. The other option begins with a product of initial states of automata and then searches only pairs which are available. *Symboliclib* uses this approach to make automata processing faster and to lower computing demands.

Chapter 8

Experimental Evaluation

This chapter describes the experimental evaluation of the algorithms for both classic finite and symbolic automata. As *symboliclib* was created for fast, easy and intuitive implementation, a comparison of an algorithm implemented in VATA and in *symboliclib* is also given. Evaluations compare execution times of VATA library with classic finite automata in and symbolic automata in *symboliclib*.

For the evaluations we used NFA from abstract regular model checking provided by VATA library. Since VATA does not support symbolic automata, the finite automata were transformed into equivalent symbolic automata so that the execution times are comparable. The evaluation was done on the set of over 20 00 pairs of NFA with a number of transitions varying from 10 to 27 000. The tests were performed on a computer with ubuntu 14.04 LTS, Intel(R) Core(TM) i3-3120M CPU (2,5 GHz, 2 cores, 256 K cache) and 4GB RAM.

As was estimated, *symboliclib* is slower than VATA library. The inefficiency comes from the differences in main goals of these libraries. VATA is created with the focus on fast and optimised automata processing, implemented in C++ which is a low-level language. The data structures used for automata storing are designed for fast search and manipulation. On the other hand, *symboliclib* is written with the focus on easy and fast learning. The data structures are not optimised for automata processing but rather for easy and intuitive manipulation. A part of the inefficiency can also be caused by the implementation language Python, which is generally slower than C++.

8.1 Intersection

For the testing of automata intersection, pairs of automata were used. The results can be seen in Figure 8.1 and Table 8.1. As estimated, *symboliclib* is slower than VATA library. The difference when working with smaller automata is not so notable for humans. For automata pairs that have around less than 5 000 transitions, the VATA execution time is 0.252 seconds, while *symboliclib* takes 1.544 seconds for classic and 3.372 seconds for symbolic automata. In these cases, using *symboliclib* for prototyping is efficient, since the result can be computed in a reasonable time that is not much higher than in optimised libraries.

With automata pairs that have over 5000 transitions together, using *symboliclib* ceases to be efficient, since the operation takes significantly higher time than in VATA. Fortunately, this does not limit the intended use of the library, since automata having less than 5000 transitions together may be sufficiently used for testing of new algorithms.

Figure 8.1: Automata intersection execution times

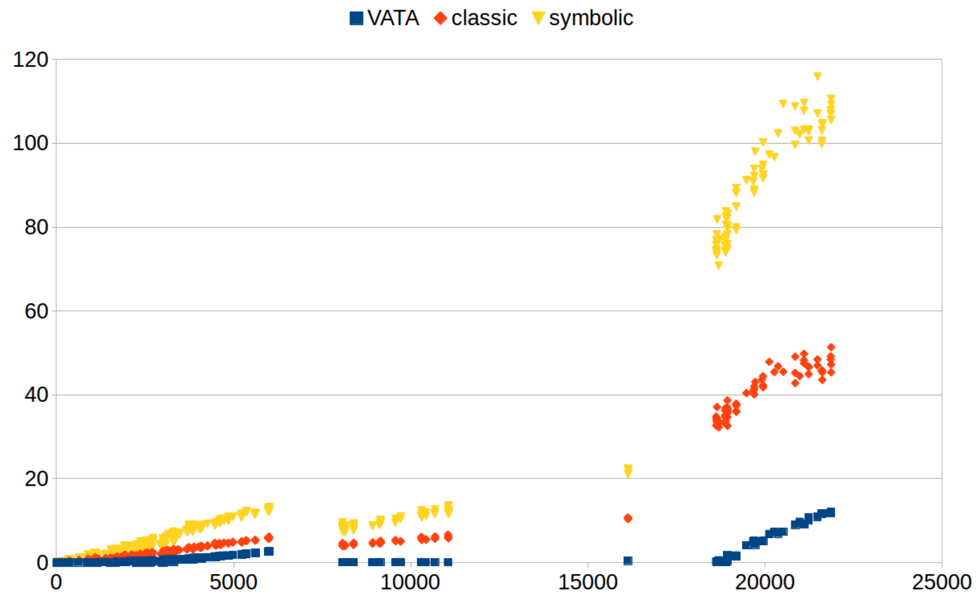


Table 8.1: Average automata intersection execution times (in seconds)

size	**vata**	**classic**	**symbolic**
0 - 5 000	0.252	1.544	3.372
5 000 - 10 000	1.116	4.869	10.221
10 000 - 15 000	0.066	5.895	11.891
15 000 - 20 000	1.663	35.270	78.516
20 000 - 25 000	8.813	45.521	102.165

The next interesting property of Figure 8.1 is the difference in classic and symbolic automata processing. We can observe that processing classic automata takes generally less time than the same operation on equivalent symbolic automata. This fact supports the theory that reimplementing some of the automata operations for classic automata increases the efficiency of the library. This is because the predicates in symbolic automata represent a set of symbols and therefore the intersection and union of such predicates include operations over sets in Python. On the other hand, classic finite automata work with symbols, where intersection and union may be replaced by simple equality which is a faster operation than operations over sets.

8.2 Language inclusion

Symboliclib supports three different algorithms for language inclusion testing described in 7.4. The Figure 8.2 shows the comparison of them as well as the execution time in the VATA library. Average inclusion times can be found in Table 8.2. Language inclusion was tested on more than 3 000 pairs of classic finite automata. The testing set included deterministic as well as nondeterministic automata, most of them having less than 3 000 transitions.

Generally, simple and classic inclusion checking should be more efficient for smaller or deterministic automata, when determinization is easy and fast or not needed at all. In

Figure 8.2: Language inclusion execution times

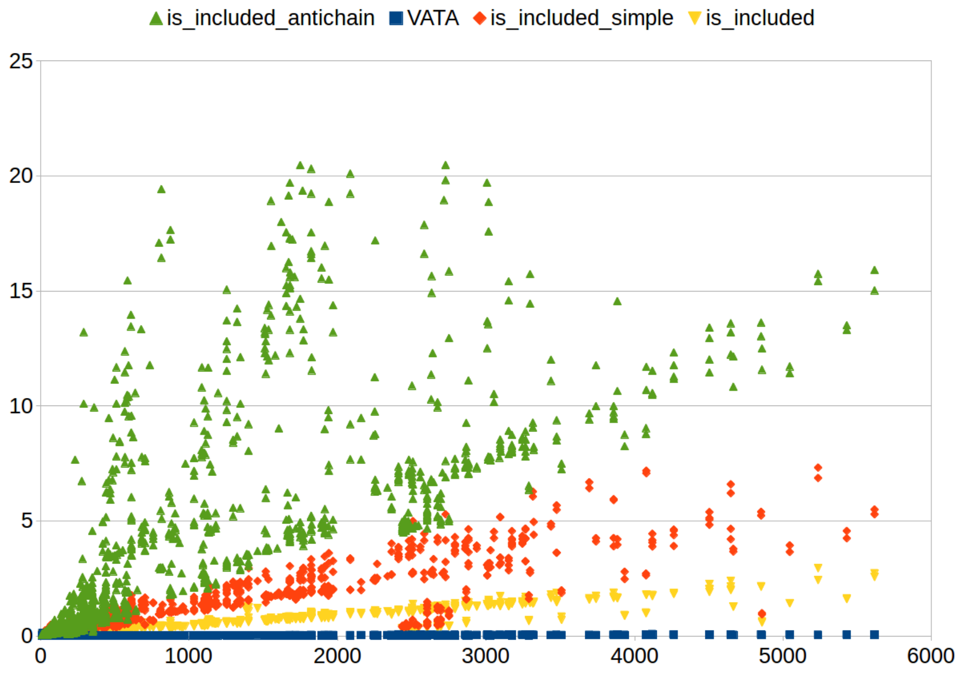


Table 8.2: Average language inclusion execution times (in seconds)

size	vata	is_included_simple	is_included	is_included_antichain
0 - 1 000	0.005	0.339	0.139	1.004
1 000 - 2 000	0.020	1.854	0.680	7.485
2 000 - 3 000	0.028	2.166	0.810	7.247
3 000 - 4 000	0.032	3.005	1.102	8.486
4 000 - 5 000	0.049	4.449	1.724	11.717
5 000 - 6 000	0.050	5.170	2.091	14.006

such cases, the computation of simulations takes much longer time than simple inclusion checking. The inclusion checking using simulations and antichains should be more efficient in nondeterministic automata that have a big number of states or transitions which can be exponentially increased during determinization.

The experimental evaluation results confirm the theory that simple and classic inclusion checking is efficient for smaller automata having less than 1 000 states. In these cases, the execution time in VATA library is 0.005 seconds and in *symboliclib* 0.23 seconds.

As for inclusion checking using the antichain algorithm, the results are not that satisfying. Antichains algorithm takes significantly longer time for automata having over 1 000 states. An average time for automata having between 1 000 and 2 000 states is 7.485 second. The median of execution times is smaller — only 4.8 seconds, but this is still much more than the algorithms using determinization. This is caused by the algorithm for computing simulations relation. For big automata, the chosen implementation of simulations is slow because of looping multiple times through the whole alphabet. A solution to this inefficiency would be to implement another algorithm for computing simulations relation.

For automata having less than 1 000 states, the antichains algorithm takes averagely 1.004 seconds, but the median is again smaller — only 0.3 seconds. As can be seen in the Figure 8.2, in some cases the antichains algorithm is faster than the algorithms using determinization. For the other automata, the determinization is easy or the automata are already deterministic, which makes the computing of simulations useless.

8.3 Programming simplicity

In this section, an example of implementation in *symboliclib* in comparison with VATA is given.

The simplicity of programming in *symboliclib* can be shown on the complement of automata. Code for this algorithm in *symboliclib* can be seen in Figure 9 and in VATA in Figure 10. Some parts of the code not important for this comparison such as comments have been skipped.

In *symboliclib*, the automaton is first determinized and then the final states are exchanged for nonfinal states through set operations. This process comes directly from the definition of language complement.

In VATA, complement requires looping through the automata states and checking whether they are final or nonfinal. The constructing of complement is less clear because of multiple nested loops.

As the *symboliclib* implementation comes directly from the definition of complement, it is easier to understand. The code from VATA is not so clear and is less user-friendly because of not making the determinization explicit. The user has to remember to determinize the automaton first and after that constructing the complement.

Algorithm 9: Automata complement algorithm in symboliclib

```
Input: NFA $self = (\Sigma, Q_A, I_A, F_A, \delta_A)$
Output: NFA $complement = (\Sigma, Q_B, I_B, F_B, \delta_B)$; $L_{complement} = \Sigma^* - L_{self}$
1  det = self.determinize();
2  complement = deepcopy(det);
3  complement.final = det.states - det.final;
4  return complement
```

Algorithm 10: Automata complement algorithm in VATA

```
Input: NFA $aut = (\Sigma, Q_A, I_A, F_A, \delta_A)$
Output: NFA $res = (\Sigma, Q_B, I_B, F_B, \delta_B)$; $L_{res} = \Sigma^* - L_{aut}$
1  ExplicitFA res;
2  for auto stateToCluster : *transitions do
3      if !aut.IsStateFinal(stateToCluster.first) then
4          res.SetStateFinal(stateToCluster.first);
5      end
6      for auto symbolToSet : *stateToCluster.second do
7          for auto stateInSet : symbolToSet.second do
8              if !aut.IsStateFinal(stateInSet) then
9                  res.SetStateFinal(stateInSet);
10             end
11         end
12     end
13 end
14 return res
```

Chapter 9

Conclusion

The goal of this thesis was to create an automata library suitable for fast prototyping of new algorithms. The library supports classic finite and symbolic automata as well as simple operations on transducers. Some state-of-the-art algorithms have been implemented including simulations relation computation and language inclusion checking using antichains.

The library was designed with the focus on easy manipulation and fast learning curve. Used data structures are transparent and simple-to-understand. The library supports easy adding new types of predicates and new algorithms.

The created library *symboliclib* was implemented in Python3 as Python is a popular language with relatively simple syntax. The input format of the library is Timbuk which has a simple intuitive syntax and is also used in other automata libraries such as VATA. *Symboliclib* implements basic operations over automata and transducers such as intersection, union, determinization or minimalization. Language inclusion checking, which is an important operation in formal verification, has been implemented using three different algorithms.

The library was tested on set of automata provided by the supervisor of this thesis. The results indicate that while the library is slower than other existing optimised libraries, it works correctly and can be used for predesigned purposes. A trade off to this inefficiency is transparent design and fast learning curve which makes prototyping and testing of new algorithms possible. *Symboliclib* is roughly 8 times slower than VATA on automata having less than 15 000 transitions but the execution time grows exponentially. This fact does not limit testing of new algorithms using the *symboliclib* library in any significant way. The best result were achieved with automata having less than 5000 transitions.

For further extending *symboliclib*, more operations over transducer could be implemented. The library could be modified to support different input and output alphabets of transducers and to support transducers with epsilon transitions.

For further optimisation of the library, more evaluation could be done to determine problem sections of implemented algorithms. These algorithms can then be optimised which will add up to the efficiency. Some state-of-the-art algorithms such as simulation relation could be further optimised or implemented in other ways. As was determined in the evaluation, operations over classic finite automata take significantly lower time than on symbolic automata so some of the operations currently implemented only for symbolic automata can be modified and implemented for classic finite automata.

As another optimisations, simulations and antichains could be altered to work faster on symbolic automata. Currently, the implementation of these operations is simple and yields correct results. On the other hand, it is not very efficient since it basically transforms symbolic automata transitions to classic finite automata. The algorithms could be designed in another way to work directly with the predicates instead of checking all symbols of automata alphabet. These modifications will contribute to making *symboliclib* faster and more efficient.

As of now, *symboliclib* is not very optimised, so it should not be used for processing automata having more than 15 000 transitions. For this purpose, other complex and optimised libraries are available. However, *symboliclib* is suitable for new automata algorithms implementation and testing or for study purposes which was the goal of this thesis.

Bibliography

[1] Web pages to FAdo library. [Online]. [Visited 10.12.2016].
Retrieved from: http://fado.dcc.fc.up.pt/

[2] Web pages to Prolog SFA library. [Online]. [Visited 11.12.2016].
Retrieved from: https://www.let.rug.nl/~vannoord/Fsa/

[3] Web pages to Timbuk. [Online]. [Visited 19.9.2016].
Retrieved from: http://people.irisa.fr/Thomas.Genet/timbuk/

[4] Web pages to VATA library. [Online]. [Visited 11.12.2016].
Retrieved from:
http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/

[5] Aziz Abdulla, P.; Chen, Y.-F.; Holík, L.; et al.: When Simulation Meets Antichains: On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata. *Proc. of TACAS 2010*. vol. 6015. 2010: pp. 158–174.

[6] Bjorner, N.; Hooimeijer, P.; Livshits, B.; et al.: Symbolic Finite State Transducers: Algorithms and Applications. [Online]. [Visited 30.9.2016].
Retrieved from: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/paper-59.pdf

[7] D'Antoni, L.: Github repository of symbolicautomata library. [Online]. [Visited 2.12.2016].
Retrieved from:
https://github.com/lorisdanto/symbolicautomata/blob/master/README.md

[8] Esparza, J.: Automata Theory: An Algorithmic Approach, Lecture notes. [Online]. [Visited 19.2.2017].
Retrieved from: https://www7.in.tum.de/~esparza/autoskript.pdf

[9] Henzinger, M. R.; Henzinger, T. A.; Kopke, P. W.: Computing Simulations on Finite and Infinite Graphs. [Online]. [Visited 15.9.2016].
Retrieved from:
https://infoscience.epfl.ch/record/99332/files/HenzingerHK95.pdf

[10] Holík, L.; Vojnar, T.: *Simulations and Antichains for Efficient Handling of Finite Automata*. Brno: Faculty of Information Technology. 2010. ISBN 978-80-214-4217-7.

[11] Hopcroft, J. E.: An n log n algorithm for minimizing the states in a finite automaton. [Online]. [Visited 10.12.2016].
Retrieved from:
http://i.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf

[12] Hopcroft, J. E.; Motwani, R.; Ullman, D. J.: *Introduction to automata theory, languages, and computation*. Brno: Boston : Addison-Wesley. 2001. ISBN 0-201-44124-1.

[13] Ilie, L.; Navaro, G.; Yu, S.: On NFA Reductions. [Online]. [Visited 15.9.2016]. Retrieved from:
`http://liacs.leidenuniv.nl/~bonsanguemm/StudSem/FI2_NFAreduct.pdf`

[14] Meduna, A.: *Automata and languages : theory and applications*. London : Springer. 2000. ISBN s81-8128-333-3.

[15] van Noord, G.; Gerdemann, D.: Finite State Transducers with Predicates and Identities. [Online]. [Visited 30.9.2016].
Retrieved from: `https://www.let.rug.nl/~vannoord/papers/preds.pdf`

[16] Veanes, M.: AutomataDotNet. [Online]. [Visited 2.12.2016].
Retrieved from: `https://github.com/AutomataDotNet/Automata`

[17] Šimáček, J.; Vojnar, T.: *Harnessing Forest Automata for Verification of Heap Manipulationg Programs*. Brno: Faculty of Information Technology. 2012. ISBN 978-80-214-4653-3.

Appendices

Appendix A

Storage Medium

The storage medium contains the sources of created library. It also contains an electronic version of this text report.