BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

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
March 12, 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

1 Introduction **5**

2 Preliminaries **7**
2.1 Languages . . . 7
2.2 Finite automata . . . 7
2.2.1 Nondeterministic finite automaton . . . 7
2.2.2 Deterministic finite automaton . . . 8
2.2.3 Run of a finite automaton . . . 8
2.2.4 Language of a finite automaton . . . 9
2.2.5 Complete DFA . . . 9
2.2.6 Well-specified DFA . . . 9
2.2.7 Minimal DFA . . . 10
2.2.8 Finite automata operations . . . 10
2.3 Regular Languages . . . 11
2.3.1 Closure Properties . . . 11
2.3.2 Decidability . . . 12
2.4 Finite transducer . . . 13
2.4.1 Translation of a finite transducer . . . 13
2.5 Predicates . . . 14
2.5.1 Operations over predicates . . . 14
2.6 Symbolic automata . . . 15
2.7 Symbolic transducers . . . 16

3 Algorithms for symbolic automata and transducers **17**
3.1 Intersection . . . 17
3.2 Union . . . 17
3.3 Determinization . . . 18
3.4 Minimization . . . 19
3.5 Optimization . . . 19
3.5.1 Satisfiability . . . 19
3.5.2 Removing useless states . . . 20

4 Effective algorithms for automata and transducers **21**
4.1 Reduction and equivalence . . . 21
4.2 Reduction and simulations . . . 22
4.3 Antichains . . . 22
4.4 Antichains and simulations for symbolic automata . . . 24

5 Existing Libraries for Automata **26**
5.1 AutomataDotNet . . . 26
5.2 symbolicautomata . . . 26
5.3 VATA . . . 27
5.4 FAdo . . . 27
5.5 FSA . . . 27

6 Design **28**
6.1 Design of the library . . . 28
6.2 Adding predicates . . . 30
6.2.1 Parsing predicates . . . 30
6.2.2 Operations over predicates . . . 30
6.3 Default predicates . . . 30

7 Implementation **31**

8 Experimental Evaluation **32**

9 Conclusion **33**

Bibliography **34**

Appendices **36**

A Storage Medium **37**

List of Figures

2.1 Nondeterministic finite automaton 8
2.2 Deterministic finite automaton 9
2.3 Nondeterministic finite transducer 14
2.4 Symbolic automaton 15
2.5 Symbolic finite transducer 16

3.1 Minimization cleanup 19

6.1 Library Design 29

List of Tables

2.1 Conjunction and of predicates 14
2.2 Disjunction of predicates 14
2.3 Complement of predicates 15

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 [14]. Further applications are hardware design, formal verification or DNA processing. Finite automata are sintactically 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 [5, 12, 14].

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 [12]. 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 [8] and antichains [9]. 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 [15], and symbolicautomata in Java by Loris d'Antoni [6]. 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[3] 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 [8] or language inclusion checking with antichains optimisation [4]. 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.[9, 11, 13, 14, 7]

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.

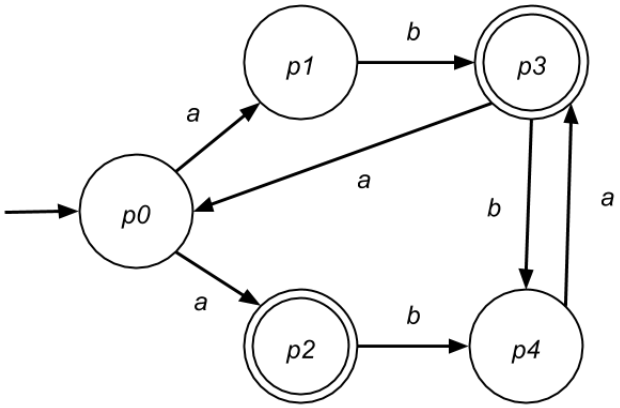
2.2 Finite automata

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.

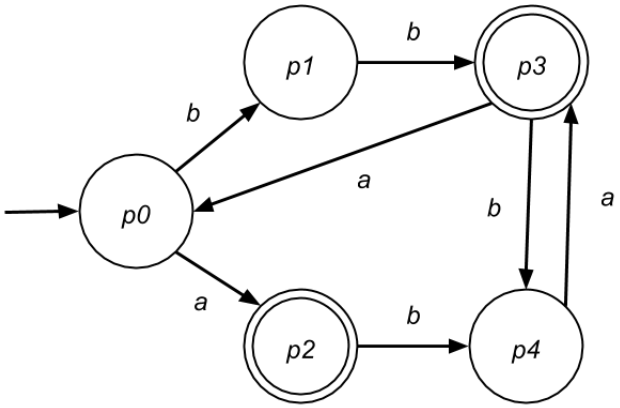
2.2.3 Run of a finite automaton

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 $L_D = L_A$

$Q_D, \delta_D, F_D \leftarrow \emptyset$;
$\mathcal{W} = \{I\}$;
while $\mathcal{W} \neq \emptyset$ **do**
    pick Q' from $\mathcal{W}$;
    add Q' to Q_D;
    if $Q' \cap F \neq \emptyset$ **then**
        add Q' to F_D;
    end
    for $a \in \Sigma$ **do**
        $Q'' \leftarrow \bigcup_{q \in Q'} \delta(q, a)$
        if $Q'' \notin Q_D$ **then**
            add Q'' to $\mathcal{W}$
        end
        add (Q', a, Q'') to δ_D
    end
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: $L = \{a_1 \dots a_n \in \Sigma^* \mid y = a_n \dots a_1 \in L\}$.
- 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 [11, 7].

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

Finitetransducers 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:

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

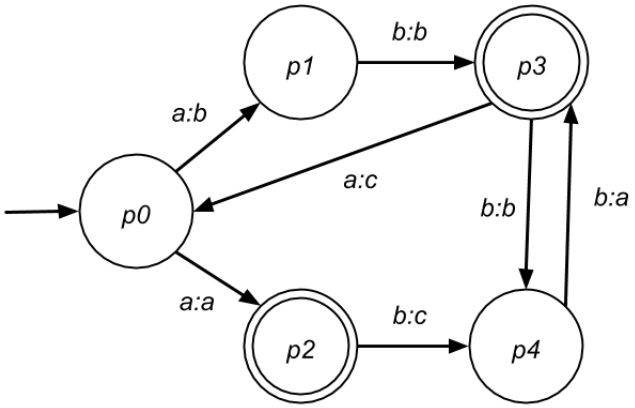
An input language corresponding to $T(M)$ is:

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

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** $\wedge$ **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** $\vee$ **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.

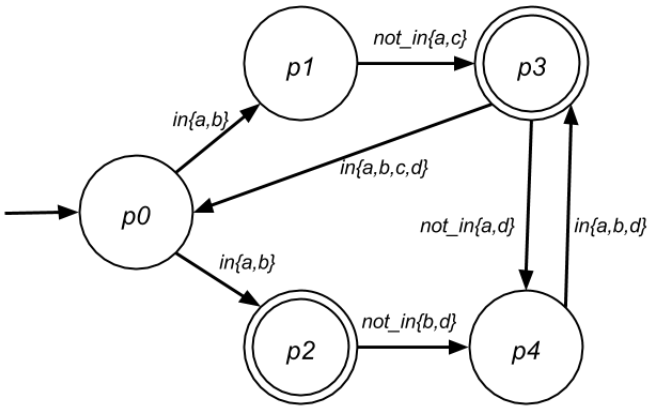
An example of SA A with predicates *in* and *not_in* introduced in 2.5 is shown in Figure 2.4.

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:

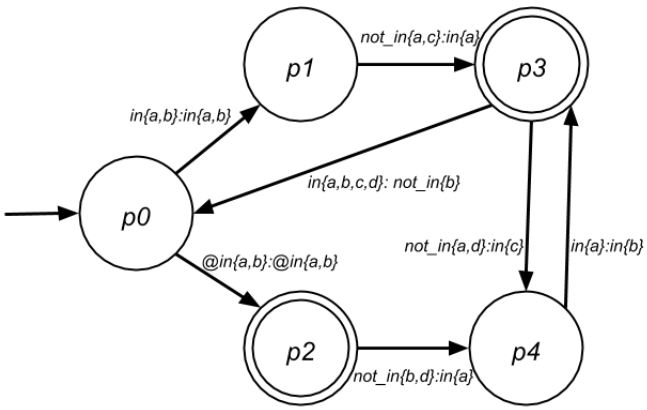
- Σ is an input alphabet
- Ω is an output alphabet
- Q is a finite set of states
- $I \subseteq Q$ is a nonempty set of initial states
- $F \subseteq Q$ is a set of final states
- Π is a set of predicates over Σ
- $\delta \subseteq Q \times (\Pi \cup \{\varepsilon\}) \times (\Pi \cup \{\varepsilon\}) \times Q$ 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. [5, 14]

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 $(d, \sigma, q) \in \delta$.[13]

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'

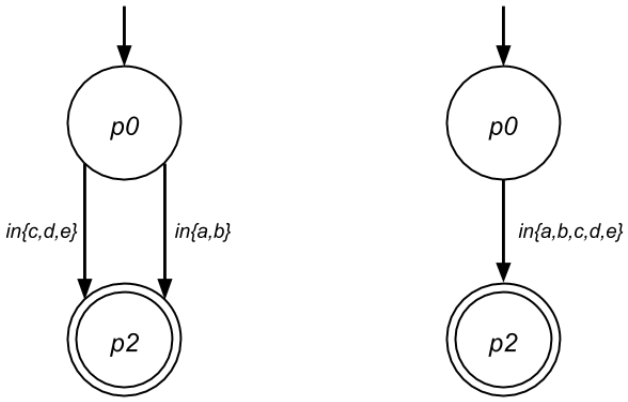
$\Pi_e \leftarrow \emptyset$;
for *each* $q \in Q'$ **do**
  $\Pi' = \{\pi | (q, \pi) \in \delta\}$;
  $C \leftarrow$ all boolean combinations of $\pi \in \Pi$;
  $\Pi_e = \Pi_e \cup C$
end
return Π_e

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 $L_D = L_A$

$Q_D, \delta_D, F_D \leftarrow \emptyset$;
$\mathcal{W} = \{I\}$;
while $\mathcal{W} \neq \emptyset$ **do**
  pick Q' from $\mathcal{W}$;
  add Q' to Q_D;
  if $Q' \cap F \neq \emptyset$ **then**
    add Q' to F_D;
  end
  $\Pi_e \leftarrow$ get exclusive predicates for Q';
  for $\pi \in \Pi_e$ **do**
    $Q'' \leftarrow \bigcup_{q \in Q'} \delta(q, \pi)$
    if $Q'' \notin Q_D$ **then**
      add Q'' to $\mathcal{W}$
    end
    add (Q', π, Q'') to δ_D
  end
end

Figure 3.1: Minimization cleanup



3.4 Minimization

In Hopcroft's minimization algorithm [10], 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

Effective 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 [8, 9, 12].

In many operations with finite automata, we need a minimised version of the automata. 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.

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 $p \xrightarrow{a} p' \in \delta$ an 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 deadend 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 modified 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.

A pseudocode for efficient simulations computation is given in Algorithm 5. 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.

4.3 Antichains

Antichains are an optimisation partly based on simulations introduced in 4.1 that can be used for universality checking and language inclusion of finite automata. The *universality problem* for an FA A is to decide whether $L(A) = \Sigma^*$. The classical algorithm firstly converts the automaton to a deterministic one through subset construction and then checks if every subset of Q is reachable. The universality checking can stop as soon as the first

Algorithm 5: Algorithm for automata reduction with simulations

Input: NFA $A = (\Sigma, Q, I, F, \delta)$
Output: $\not\preceq_R$

```
1  for each q ∈ Q and a ∈ Σ do
2  |  compute δ^r(q, a) as an linked list;
3  |  compute card(δ(q, a));
4  end
5  initialize all N(a)s with 0s;
6  ω := ∅;
7  C := NEW_QUEUE();
8  for each f ∈ F do
9  |  for each q ∈ Q − F do
10 |  |  ω := ω ∪ {(f, q)};
11 |  |  ENQUEUE(C, (f, j));
12 |  end
13 end
14 while C ≠ ∅ do
15 |  (i, j) := DEQUEUE(C) ;
16 |  for each a ∈ Σ do
17 |  |  for each k ∈ δ^r(j, a) do
18 |  |  |  N(a)_ik := N(a)_ik + 1;
19 |  |  |  if N(a)_ik == card(δ(k, a)) then
20 |  |  |  |  for l ∈ δ^r(i, a) do
21 |  |  |  |  |  if (l, k) ∉ ω then
22 |  |  |  |  |  |  ω := ω ∪ {(l, k)};
23 |  |  |  |  |  |  ENQUEUE(C, (l, k))
24 |  |  |  |  |  end
25 |  |  |  |  end
26 |  |  |  end
27 |  |  end
28 |  end
29 end
```

contradiction is found. This approach is very inefficient since the determinization can exponentially increase the number of states of the given FA. Then, if the automaton is universal, we are forced to check every subset of states.

The language inclusion checking algorithm is also trying to find a contradiction. For this algorithm, various optimisations are available. A pseudo code for language inclusion can be found in Algorithm 6.

First optimisation 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 11–14 in the pseudo code. The basic explanation is that if the algorithm encounters a macrostate R (*macrostate* is defined as a subset of states from Q) 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 number of accepted words.

The second 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 $p_2 \in P$, and $p_1 \preceq p_2$. 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 algorithm also reduces the number of macrostates that need to be checked and therefore the used memory space and computing capacity.

4.4 Antichains and simulations for symbolic automata

Antichains for symbolic automata can be computed by the algorithm desribed in Algorithm 6 since this algorithm does not work directly with the alphabet of automata. The difference would be in the implementation of simulations checking.

In simulations computation, we may reuse the principle introduced in 3.3. Instead of considering a pair of state and symbol, we consider a pair of state and its exclusive predicate. To get all exclusive predicates, we must create all boolean combinations of predicates leaving a given state. This change would modify lines 1 and 16 of Algorithm 5:

1. For(each $q \in Q$ and $\pi \in \{$ exclusive predicates of $q\}$)
16. For(each $\pi \in \{$ exclusive predicates of $j\}$)

For more efficient approach we may implement an optimisation from 3.5 and loop only through satisfiable predicates.

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) | i ∈ I_A} then
2  |  return FALSE;
3  end
4  Processed:=∅;
5  Next:= {(s, Minimize(I_B)) | s ∈ I_A};
6  while Next ≠ ∅ do
7  |  Pick and remove a product-state (r, R) from Next and move it to Processed;
8  |  foreach (p, P) ∈ {(r', Minimize(R')) | (r', R') ∈ Post(r, R)} do
9  |  |  if (p, P) is a witness product-state then
10 |  |  |  return FALSE;
11 |  |  else
12 |  |  |  if ∄p' ∈ P s.t. p ⪯ p' then
13 |  |  |  |  if ∄(x, X) ∈ Processed ∪ Next s.t. p ⪯ x ∧ X ⪯^{∀∃} P then
14 |  |  |  |  |  Remove all (x, X) from Processed ∪ Next s.t. x ⪯ p ∧ P ⪯^{∀∃} X;
15 |  |  |  |  |  Add (p, P) to Next;
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  end
20 end
21 return TRUE;
```

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 [15] 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 [6] 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 [3, 16] 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 process of adding new types of predicates is explained.

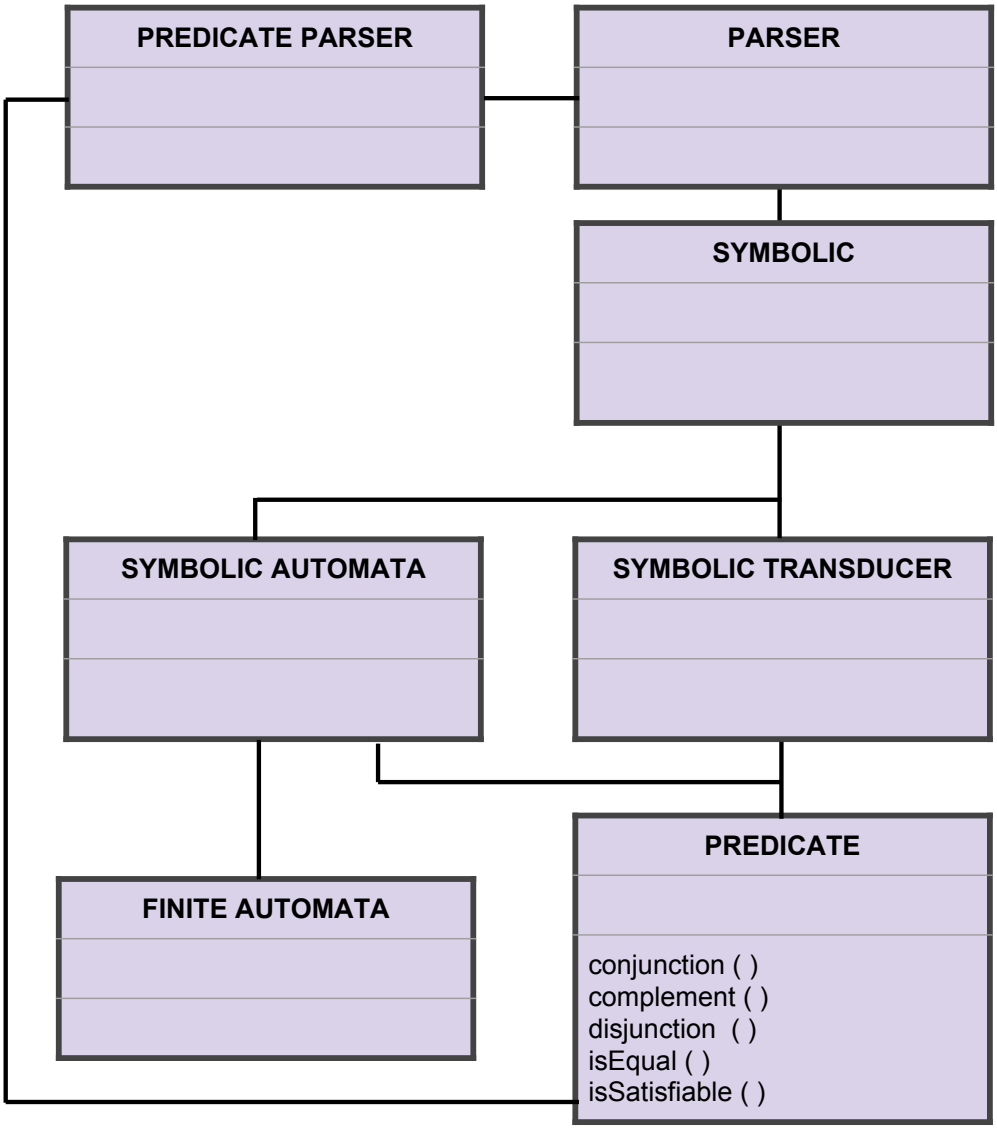
The library is designed to efficient manipulation with symbolic automata and transducers. Basic operations are available for each supported type of automata. Adding new operations should be easy and the library will be used for fast prototyping of new algorithms. None of the operations should be dependent 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 independent parts, 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 an `Predicate` object which respects the defined interface. 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 transitions as an oriented graph such as intersection or union.
- `Symbolic automata` is a class for operations over symbolic automata. It contains an attribute `deterministic` which indicates, whether we are working with a deterministic automata or a nondeterministic one. It includes basic operations such as union, intersection or determinization as well as efficient and advanced ones, such as simulations and antichain.
- `Symbolic Transducer` contains implementations of operations over symbolic transducers.
- `Finite automata` is a class in which some algorithms are optimized for classical finite automata. Classical finite automata operations do not need to work with union or

Figure 6.1: Library Design



intersection of symbols, only simple equality is required. Therefore some of the algorithms implemented in `Symbolic automata` may be modified to work more efficiently when dealing with classic symbols instead of predicates.

A reason for this design is to allow the user to provide own predicate types. When adding new predicate type, only providing a predicate parser and a predicate class that implements all demanded operations is mandatory.

6.2 Adding predicates

When adding predicates, predicate class and parser must be provided. Predicate class must implement the interface defined in 6.2.2

6.2.1 Parsing predicates

Every time the parser finds a transition label, it calls Predicate Parser and stores the returned value. This allows simple extension of supported predicates. All the user needs to do in terms of parsing is to provide a parser for the predicate type.

6.2.2 Operations over predicates

The implemented algorithms require certain operations over predicates. Therefore the class of the predicate needs to implement the defined interface.

The operations needed are:

- union
- intersection
- complement
- satisfiability

Union and *intersecton* are needed for basic algorithms such as determinism checking or uniting transitions between the same pair of states. Algorithms such as determinisation and minimisation also need *complement* to be implemented. $Satisfiability$ is used in optimisations described in 3.5.

6.3 Default predicates

Predicates provided by the library as default are:

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

Chapter 7

Implementation

Chapter 8

Experimental Evaluation

Chapter 9

Conclusion

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

[4] 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.

[5] 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

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

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

[8] 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

[9] 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.

[10] 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

[11] 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.

[12] 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

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

[14] 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

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

[16] 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.