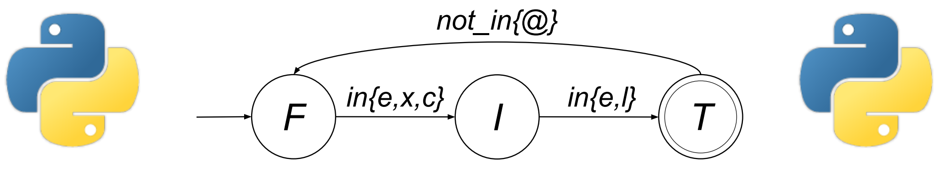
Excel@FIT 2017
http://excel.fit.vutbr.cz

Symbolic Automata Library for Fast Prototyping

Michaela Bieliková



Abstract
Finite state automata are widely used in the fields of computer science such as formal verification, system modelling, and natural language processing. However, the models representing the real systems are complicated and may be defined upon big alphabets. In such cases, using classical finite state automata is not efficient and more concise representation is needed. So called symbolic automata are suitable formalism for this purpose since they employ predicates as transition labels. One predicate represents set of symbol for which condition hold. Currently no library that allows easy and intuitive manipulation with symbolic automata exists. As a result of this work, a Python library *symboliclib* was created. It supports symbolic automata with different types of predicates as well as classic finite automata working with symbols. The library is slower than more complex ands optimised automata libraries when working with big automata. The inefficiency is mainly caused by using Python which is a high level language slower than lower-level languages such as C or C++. A trade off to this inefficiency is the simplicity and fast learning curve of use of the library.

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

Supplementary Material: Github Repository

*xbieli06@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Finite automata have syntactically simple but general formalism. They 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 [1]. Another application is formal verification for example to model reachable configuration of the analysed system. Finite transducers are used in formal verification to model behaviour of an analysed system. They can be also used in natural language processing where they can describe phonological rules or form translation dictionaries [1, 2].

While finite automata are of practical use, when large alphabets are used the computing demands quickly increase. In practice, big alphabets are 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 double. This number further increases in alphabets that have a symbol for every syllable such as Chinese or Japanese or applications in which the symbols are words such as electronic dictionaries which often have more than 200K words [2]. In these cases, an modification of automata with predicates replacing elementary symbols can be used. Predicates allow representing a set of symbols by 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 generalisable to symbolic automata and transducers.

Important operations over automata and transducers, such as minimisation and language inclusion (often used in formal verification) need the automata to be determinised first, which can exponentially increase the number of states or transitions. Fortunately, advanced algorithms that allow inclusion checking without determinisation, exist. This paper employs two heuristics for language inclusion checking algorithms based on simulations and antichains [3]. Simulations is the language preserving relation of each pair of states and then eliminate the states which have the same behaviour for every possible input symbol. Antichains study the relations of sets of automata states.

Currently there are libraries for symbolic automata. Most of these libraries are either complex and optimised or immature and still a prototype. The optimised libraries have often complex inner representation of automata designed with focus on efficient automata processing which makes the representation less understandable for humans and therefore implies a slow learning curve. This can slow down designing new algorithms because of the time needed to study the inner automata representation.

Therefore, the goal of this work 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. The library should also include some advanced algorithms for automata such as already mentioned language inclusion checking employing simulations and antichains.

2. Theoretical background

This chapter contains theoretical background for this paper. First, languages and classic finite automata will be defined, then predicates are introduced. After the definition of predicates, symbolic automata and transducers are described. Proofs are not given but can be found in the referenced literature [1, 4].

2.1 Languages

Let Σ be an *alphabet* — finite, nonempty set of symbols. Common examples of alphabets include the binary alphabet $\Sigma = \{0,1\}$ 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 and their concatenation is 00111.

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

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

2.2 Finite automata

A *nondeterministic finite automaton* (NFA) is a tuple $A = (\Sigma, Q, I, F, \delta)$, 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
- $\delta \subseteq Q \times \Sigma \times Q$ is a 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 over the label a.

A *deterministic finite automaton* (DFA) is a special case of an NFA, where $|I| = 1$ and δ is a partial function restricted such that $p \xrightarrow{a} q \wedge p \xrightarrow{a} q' \Rightarrow q = q'$. Informally said, a DFA must have exactly one initial state and cannot contain more transitions from one state over the same symbol.

2.3 Finite transducer

Finite transducers differ from finite automata in transition labels. While finite automata labels contain only one input symbol, both input and output symbols are presented in finite transducers. Thus, transducers have two alphabets — one consists of input symbols and one of output symbols. Finite transducers can be informally described as translators which for a symbol in input word generate a symbol in an output word.

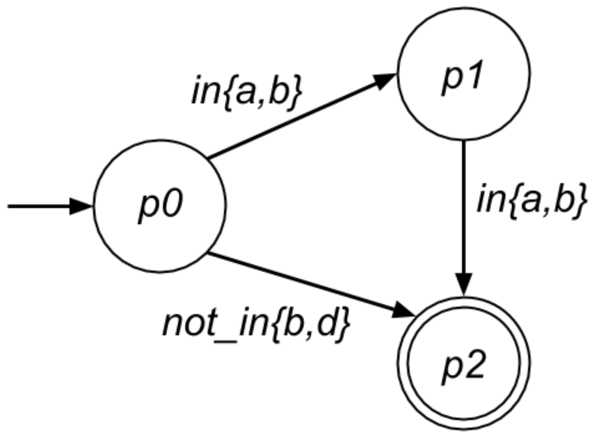
The transitions relation for finite transducers is defined as $\delta \subseteq Q \times (\Sigma : \Omega) \times Q$. We use $p \xrightarrow{a:b} q$ to denote the transition from a state p to the state q reading the input symbol a and generating the output symbol b.

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

2.4 Predicates

A *predicate* π is a formula representing a subset of Σ. Π denotes a given set of predicates such that, for

Figure 1. Symbolic automaton



each element $a \in \Sigma$ there is a predicate representing $\{a\}$ (e.g. a or $in\{a\}$), and Π is effectively closed under Boolean operations.

Predicates provided by the library by default are inspired by a Prolog library FSA [5]. Predicate $in\{a_1,\ldots,a_i\}$ represents subset $\{a_1,\ldots,a_i\} \subseteq \Sigma$ and predicate $not\_in\{a_1,\ldots,a_i\}$ represents subset $\Sigma - \{a_1,\ldots,a_i\}$.

2.5 Symbolic automata

A *symbolic automaton* A is a tuple $A = (\Sigma, Q, I, F, \Pi, \delta)$, where Π is a set of predicates over Σ and the transition relation is defined as $\delta \subseteq Q \times (\Pi \cup \{\varepsilon\}) \times Q$. An example of SA can be found in Figure 1.

Every SA with a finite alphabet can be transformed into an NFA by removing Π and transforming each transition into an equivalent set of transitions in which the predicate is transformed to the corresponding elementary symbols of Σ. Since every SA can be transformed into NFA, 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 the most of the algorithms for finite automata can be modified to work on symbolic automata.

2.6 Symbolic transducers

A *symbolic transducer* A is a tuple $A = (\Sigma, \Omega, Q, I, F, \Pi, \delta)$, where Ω is an output alphabet and the transition relation is $\delta \subseteq Q \times (\Pi \cup \{\varepsilon\}) \times (\Pi \cup \{\varepsilon\}) \times Q$.

A special case of a transition is *identity*. Identity takes an input a and copies it on the output. In *symboliclib*, identity is denoted by @ in front of the predicate (e.g. $@in\{a,b\} : @in\{a,b\}$).

As well as symbolic automata, every symbolic transducer can be transformed into classic finite transducer. Therefore every operation applicable to finite transducer can be applied on symbolic transducer after the transformation, but the transformation can usually be avoided by generalisation of algorithms.

3. Efficient algorithms for automata and transducers

In many operations with finite automata, we need a minimised version of the automata. However, a textbook approach to minimisation of NFA includes determinisation which may lead to the state explosion. Therefore more efficient methods for NFA reduction are used which avoid the full determinisation. In this work we use algorithms based on simulations and antichains. A reduction based on simulations usually yields an automaton that is smaller than minimal DFA but not deterministic.

The simulation relation $\preceq_R \subseteq Q \times Q$ is defined as:

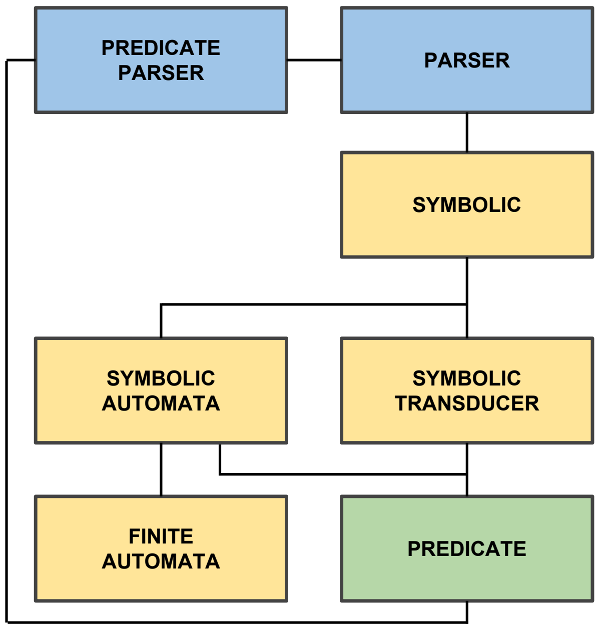
- $\preceq_R \cap (F \times (Q - F)) = \emptyset$
- for any $p, q \in Q, a \in \Sigma, (p \preceq_R q \Rightarrow \forall p' \in \delta(p,a), \exists q' \in \delta(q,a), p' \preceq_R q')$

We can merge two states p and q when $p \preceq_R q$ and $q \preceq_R p$, which implies that they are language equivalent.

The algorithm based on antichains starts searching for a final state of the product automaton of the product automaton of two automata A (the smaller one) and B (the bigger one) while pruning out the states which are not necessary to explore. We can stop the search for a pair (p, P) (where $p \in Q_A$ and $P \subseteq Q_{B_{det}}$) if there exists some already visited pair (r, R) such that $p \preceq r \wedge R \preceq P$ or if there is $p' \in P$ such that $p \preceq p'$. The main idea behind this is that if we have not found a counterexample in a small set of automaton states, there is no 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.

The second optimisation is based on simulations. It comes from observation that $L(A)(P) = L(A)(P \setminus \{p_1\})$ if there is some $p_2 \in P$, and $p_1 \preceq p_2$. Since P and $P \setminus \{p_1\}$ have the same language, if a word is not accepted from P, it is not accepted from $P \setminus \{p_1\}$ either. Also, if all words from Σ^* are accepted from P, they are also accepted from $P \setminus \{p_1\}$. Therefore, it is safe to replace the macrostate P with macrostate $P \setminus \{p_1\}$. This algorithm also reduces the number of macrostates that need to be checked and therefore the used memory space and computing capacity.

Figure 2. Library Design



4. Existing libraries

Currently, there are libraries that deal with some form of symbolic automata. AutomataDotNet in C# by Margus Veanes [6], and symbolicautomata in Java by Loris d'Antoni [7] are efficient and offer many advanced algorithms. Unfortunately, they are very complex, have a slow learning curve, and implementation of a new algorithm requires dealing with a complex internal representation of automata. Therefore are not suitable for quick prototyping of new algorithms. Then there is the VATA library[8, 9] 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 or language inclusion checking with antichains optimisation [3]. It can also handle tree automata. It is modular and therefore easily extendable, but since it is written in C++, prototyping in VATA is not easy. Moreover, VATA does not support symbolic automata. FAdo library [10] is a library for finite automata and regular expressions written in Python. Unfortunately, it is still immature and not well documented. Efficient library is FSA written in Prolog [5]. FSA offers determinisation, minimisation, epsilon removal and other algorithms for symbolic automata, and also supports transducers. Unfortunately, Prolog is not so widely used and the library is outdated.

5. Design of the library

The library consists of multiple independent parts, as is shown in Figure 2. 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 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 determinisation and minimisation as well as efficient and advanced ones, such as simulations and antichains.
- `Symbolic Transducer` contains implementations of operations over symbolic transducers.
- `Finite automata` is a class containing algorithms optimised for classical finite automata. Classical finite automata operations do not need to work with union or intersection of predicates, 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.

The library provides some predefined predicates as default:

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

All algorithms are independent from the type of predicates used in automata. However, they require certain operations over predicates. When adding a new predicate type, all of these operations must be implemented for the library to work correctly. The needed operations are:

- conjunction
- disjunction

- negation
- satisfiability
- has_letter - decide whether a symbol $a \in \Sigma$ belongs to the set defined by the predicate

6. Implementation

Symboliclib is a library implemented in Python 3. The default input format of *symboliclib* is Timbuk [11] which is so far the only supported format. For symbolic automata and transducers the Timbuk format was slightly modified so that the main principles of Timbuk format are preserved. This allows simple automata parsing which is very similar for finite automata and the symbolic ones. *Symboliclib* also supports serialisation of automata back to the Timbuk text format.

Symboliclib supports automata with epsilon rules as well as automata with multiple initial states. Most of the automata operations implemented in the library (intersection, union, determinisation, minimisation, epsilon rules removing, etc.) are based on [4].

Attributes of the automata such as alphabets and intial, 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.

For easy manipulation with automata, transitions are saved in a list of dictionaries. The key of the dictionary is a left-handend 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:

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

7. Conclusions

The goal of this work was to design and implement a library aiming to fast prototyping of new algorithms.

The library was designed and implemented and it includes mentioned state-of-the-art algorithms. These algorithms were modified to work on symbolic automata instead of classic finite automata. Two different types of predicates are supported by default.

Straightforward design of the library provides a fast learning curve. Because of implementation in high-level language Python implementation of new algorithms may be accomplished in significantly lower time that in the existing libraries.

8. Future work

An experimental evaluation of *symboliclib* could be done for further optimisations. The evaluation should include comparison with other libraries to determine its efficiency. The main focus of evaluation should be language inclusion checking which is often used in formal verification.

Some experiments could be done to determine whether simplifying algorithms for finite automata is efficient. This could be accomplished comparing the symbolic and finite automata algorithms which are implemented separately in the library such as intersection, union or language inclusion checking based on simulations.

Acknowledgements

I would like to thank my supervisor Ing. Martin Hruška for his help.

References

[1] Gertjan van Noord and Dale Gerdemann. Finite State Transducers with Predicates and Identities. [Online]. [Visited 30.9.2016].

[2] Lucian Ilie, Gonzalo Navaro, and Sheng Yu. On NFA Reductions. [Online]. [Visited 15.9.2016].

[3] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When Simulation Meets Antichains: On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata. *Proc. of TACAS 2010*, 6015:158–174, 2010.

[4] Javier Esparza. Automata Theory: An Algorithmic Approach, Lecture notes. [Online]. [Visited 19.2.2017].

[5] Web Pages to Prolog SFA library. [Online]. [Visited 11.12.2016].

[6] Margus Veanes. AutomataDotNet. [Online]. [Visited 2.12.2016].

[7] Loris D'Antoni. Github Repository of Symbolicautomata Library. [Online]. [Visited 2.12.2016].

[8] Web Pages to VATA library. [Online]. [Visited 11.12.2016].

[9] Jiří Šimáček and Tomáš Vojnar. *Harnessing Forest Automata for Verification of Heap Manipulationg Programs*. Faculty of Information Technology, Brno, 2012.

[10] Web Pages to FAdo library. [Online]. [Visited 10.12.2016].

[11] Web Pages to Timbuk. [Online]. [Visited 19.9.2016].